Microsoft Dynamics NAV -selainratkaisu

HAMK
HÄMEEN AMMATTIKORKEAKOULU

Ammattikorkeakoulututkinnon opinnäytetyö

Tietojenkäsittelyn koulutusohjelma

Visamäki, 20.12.2011

Mikko Marttila

HAMK
HÄMEEN AMMATTIKORKEAKOULU

OPINNÄYTETYÖ

Tietojenkäsittelyn koulutusohjelma
Hämeenlinna

Työn nimi	Microsoft Dynamics NAV -selainratkaisu
Tekijä	Mikko Marttila
Ohjaava opettaja	Lasse Seppänen
Hyväksytty	_____._____.20_____
Hyväksyjä	

HAMK
HÄMEEN AMMATTIKORKEAKOULU

VISAMÄKI
Tietojenkäsittelyn koulutusohjelma

Tekijä Mikko Marttila **Vuosi** 2011

Työn nimi Microsoft Dynamics NAV -selainratkaisu

TIIVISTELMÄ

Microsoft Dynamics NAV -selainratkaisu on Hämeen ammattikorkeakoulun toimeksiantoon perustuva opinnäytetyö. Työssä tutkitaan .NET-teknologiaan perustuvien selainratkaisuiden resurssitehokkuutta. Selainratkaisut myös integrointiin kohteenaan Microsoft Dynamics NAV ja sen kautta jaettava SQL Server -tietokanta. Toimeksiantoon perustuen projektissa on toteutettu kaksi kappaletta esimerkkiselainratkaisuja, jotka on testattu kuormitus- ja rasitustestauksen avulla. Kyseenomaiset testit on suunniteltu tätä projektia varten ja niiden tarkoituksena on tuottaa tietoa siitä, kuinka kyseenomaiset selainratkaisut käyttäytyvät perustasossa ja kuormituksen kasvaessa. Samalla selvitetään kuinka tutkittuja tekniikoita hyväksikäyttäen resurssitehokas sovellus on järkevää toteuttaa.

Tutkimus on kvalitatiivinen. Tutkittavaan kohteeseen on perehdytty alan asiantuntijoiden luoman taustamateriaalin kautta, jota hyödyntäen on luotu tietopohja selainratkaisujen luontia ja testauksen suunnittelua sekä testauksen toteuttamista varten.

Opinnäytetyö ja toimeksiantoon perustuva tutkimus rajautuvat seuraavalla tavalla. Työ käsittelee ASP.NET-sovelluksia, joiden integraatiota Microsoft Dynamics NAV'n kautta jaettavaan SQL Server -tietokantaan tutkitaan. Selaimena on Internet Explorer. Työ on toteutettu resurssitehokkuuden näkökulmasta. Testaus on suoritettu käytettävissä ollutta virtuaalipalvelinympäristöä mukaillen. Lopputuloksena saadaan selvitys siitä kuinka kyseenomaiset selainratkaisut on järkevää toteuttaa kahden testattavan selainratkaisun kautta saatujen tutkimustulosten mukaan. Tulokset ja selainratkaisut luovutettiin toimeksiantajalle raportin valmistuessa.

Avainsanat ASP.NET, integrointi, Microsoft SQL Server, Microsoft .NET, Visual Studio

Sivut 36 s. + liitteet 17 s.

HAMK
UNIVERSITY OF APPLIED SCIENCES

VISAMÄKI
Degree Programme in Business Information Technology

Author Mikko Marttila **Year** 2011

Subject of Bachelor's thesis Microsoft Dynamics NAV Web Browser Solution

ABSTRACT

Microsoft Dynamics NAV Web Browser Solution is a thesis which was based on the assignment ordered by HAMK University of Applied Sciences. A research was ordered for performance issues of .NET based web browser solutions. During this assignment two web browser solutions were created. Their resource efficiency and performance are tested and compared using load testing and stress testing methods. Testing was based on ASP.NET performance testing. Those tests were developed for this actual project. The web browser solutions created during this project were integrated with SQL Server database which is in this case served by Microsoft Dynamics NAV.

The researching method of this thesis is qualitative research. That means the target researched is approached from the view of ASP.NET development theory based on knowledge collected from experts views. The material is used as background material which is used in the development process of this assignment when developing ASP.NET web browser solutions and when creating the performance testing in this specific situation.

The assignment is limited into performance research. That is one important aspect when creating web browser solutions. Other limitations in this project are that it handles ASP.NET only, ASP.NET integrations with SQL Server database served by Microsoft Dynamics NAV only and the web browser used is Internet Explorer. The tests were made for a server platform which is allocated into virtual servers of HAMK University of Applied Sciences.

As a result there will be a report which gives data about how web browser solutions can be made with performance efficiency in the case similar with this one. With the report, the assignments client will get the test data report and web browser solutions created during this project.

Keywords ASP.NET, integration, Microsoft SQL Server, Microsoft .NET, Visual Studio

Pages 36 p. + appendices 17 p.

Sanasto

ADO.NET	Yleinen tiedonhallintatapa .NET-sovelluskehityksessä.
ASP.NET Cache API	ASP.NET välimuistin rajapinta.
client	Asiakas, joka on yhteydessä sovellukseen tai palvelimeen.
data	Tieto.
domain	Toimialue.
domain controller (DC)	Toimialuetta hallinnoiva tietokone.
IIS	Internet Information Services. Internet-palvelin, jolla jaetaan internet- sivuja domainilta verkkoon.
ekstranet	Verkkopalvelu, johon kirjaudutaan verkon kautta esimerkiksi internet- selainta käyttäen.
integraatio	Kahden tai useamman kohteen yhdistäminen toisiinsa.
kaatumispiste	Piste, jossa järjestelmä lakkaa vastaamasta pyyntöihin ja sulkeutuu.
konfiguraatio	Tarkoitetaan fyysisessä ympäristössä tai virtuaalipalvelimella sijaitsevaa palvelinrypästä, jotka kuuluvat samaan verkkoon.
LINQ	Language Integrated Query. Uusi .NET- teknologia versiossa 3.5.
Microsoft Dynamics NAV	Microsoftin toteuttama talouden- ja toiminnan ohjausjärjestelmä.
palvelunestohyökkäys	Siinä kohdepalvelin tai muu kohde jumitetaan toistuvalla ja runsaslukuisella data-kuormalla. Aikeenaan hidastaa tai lamauttaa kohde.

pullonkaula	Ohjelmistossa oleva kohta, joka haittaa resurssitehokasta suoriutumista.
RAM- muisti	Tietokoneen työmuisti.
SiteMap	ASP.NET -sovelluksissa käytössä oleva navigointijärjestelmä.
sovelluskehitin	Ohjelmoinnin apuväline. Tässä työssä käytössä Visual Studio.
SQL	Structured Query Language. Tietokantakyselykieli.
SQL Server	Microsoftin tietokantasovellus.
Stored Procedures	Tietojenkäsittely tekniikka ASP.NET- sovelluksissa. Käytetään SQL- komentojen käsittelyssä.
TCP/IP	Transmission Control Protocol / Internet Protocol: usean internet-liikennöinnissä käytettävän tietoverkkoprotokollan yhdistelmä.
toimialue	Alue, jossa ryhmä tietokoneita on yhdistetty toisiinsa, jotta ne voivat asioida keskenään.
työkuormaprofiili	Kartoitus, jonka avulla luodaan työkuormamalli.
Visual Studio	Sovelluskehitin, jolla voidaan luoda ja testata .NET-koodia.
verkkokapasiteetti	Määrittää verkon kyvyn suoriutua pyynnöistä.
verkkopalvelu	Sovellus, joka on jaossa palveluna verkon välityksellä.

SISÄLLYS

1. JOHDANTO

Verkkopalvelut ovat tärkeä osa nykyistä yhteiskuntaa. Monet käyttävät niitä päivittäin. Verkkopalveluja voidaan luoda monenlaisiin tilanteisiin ja niitä voidaan luoda monella tapaa. Myös sovellusten väliset integraatiot ovat nykypäivää ja ne kuuluvat yleisenä osana verkkopalveluiden sovelluskehitysprosessiin. Yksi vaihtoehto tehdä verkkopalvelu on tehdä palvelu ASP eli Active Server Pages -sivuin. Tässä opinnäytetyössä perehdytään ASP.NET-sovelluksiin, jotka integroidaan Microsoft Dynamics NAV -järjestelmään.

Opinnäytetyön lähtökohtana on Hämeen ammattikorkeakoulun Tietojenkäsittelyn koulutusohjelman toimeksianto, jossa pyydettiin tutkimusta resurssitehokkuuteen vaikuttavista tekijöistä ja resurssitehokkuuden mittaamisesta Microsoft Dynamics NAV:iin integroitavien, .NET- teknologiaa hyödyntävien, sovellusten luontiprosessissa. Tutkimuksessa minulla oli suhteellisen vapaat kädet, mutta kuitenkin muutamia reunaehtoja oli(Liite 1). Päätin toteuttaa kaksi kappaletta ekstranet-tyyppisiä selainratkaisuja, jotka on luotu ASP.NET 3.5 -yhteensopiviksi.

Selainratkaisujen toteutustekniikaksi valittiin ASP.NET 3.5. ASP.NET 3.5 oli uusi versio projektin alkuvaiheessa ja se on käytössä yhä. ASP.NET-kehitys on siirtymässä .NET 4.0 -versioon, mutta edelleen useat sovellukset hyödyntävät ASP.NET 3.5 -versiota. Työ etenee teorian kautta käytännön projektin tarkasteluun.

2. .NET FRAMEWORK JA SUORITUSKYKYTEHOKAS ASP.NET

Verkkosovellusten luominen vaatii aina oman aikansa. Toisella tekniikalla aikaa voi kulua enemmän kuin toisella tekniikalla toteuttaen. Myös työn laajuus vaikuttaa käytettyyn aikaan. Verkkosovellusta luotaessa valintamahdollisuuksia on monia. Tässä työssä käsittelyssä olevat .NET Framework ja ASP.NET-sovellukset ovat suhteellisen helppoja rakentaa, sillä ASP.NET-kehitystyö on luonteeltaan kehittäjäystävällinen. Siltikään toimiva sovellus ei vielä itsessään ole sataprosenttisen valmis. Vaikka täydellisyyteen ei välttämättä päästä koskaan, on siihen hyvä pyrkiä. Valmiin sovelluksen jatkotoimia (joskin nämä toimet voidaan aloittaa myös kesken luontiprosessin) ovat resurssikuormitukseen perehtyminen ja resurssikuormituksen minimointi, vähintäänkin vähentäminen. Tätä toimenpidettä kutsutaan optimoinniksi. On kuitenkin syytä huomioida, että optimointi on jokaisessa tapauksessa yksilöllinen toimenpide. Aina samaa ratkaisumallia ei voida hyödyntää toisissa tilanteissa sellaisenaan, toisinaan tämäkin on mahdollista. Kyse ei siis ole absoluuttisista totuuksista, vaan absoluuttiseen tulokseen pyrkimiseen liittyvistä asioista ja toimista. On myös syytä huomioida, ettei vaikuttaviin tekijöihin kuulu vain kirjoitettu koodi, vaan optimointiprosessi on monen osatekijän summa. Kaikki tekijät eivät siis välttämättä edes ole suoranaisessa tekemisessä itse sovelluksen kanssa. Työn perehtymiskulman vuoksi keskittymiskohteena on kuitenkin ASP.NET-sovellukset ja niihin liittyvä resurssikuormituksen hallinta.(Howard 2005)

2.1 .NET Framework ja sen tärkeimmät osat

.NET Framework on Microsoftin kehittämä konsepti, jonka tarkoituksena on ollut helpottaa ohjelmistokehityksen rutiineja ja saada silti aikaan toimivia sovelluksia. Microsoft .NET Framework -sovelluskehityksessä voidaan käyttää noin kahtakymmentä ohjelmointikieltä. Niistä yleisimpiä ovat Visual Basic ja C#. Muita mahdollisia kieliä ovat esimerkiksi J# (Java), COBOL, Pascal, Perl ja Python. Koodi voidaan kirjoittaa millä tahansa niistä kielistä, joita .NET Framework tukee. Koodi voi koostua myös useammista niistä, sillä ne kulkevat sisäisen kääntäjän kautta, muuttuen yleiskieleksi. Näin ollen koodi on samassa muodossa, oli kieli mikä tahansa tuettu ja siksi ymmärrysongelmia ei tule. Jokaisella ohjelmointikielellä on kuitenkin oma kääntäjänsä. Osan niistä on toteuttanut Microsoft, osan muut tahot.(Richter 2003, 4.)

.NET Framworkin muodostuu kahdesta osasta. Ne ovat kaikille ohjelmointikielille yhtenäinen Common Language Runtime (CLR) -ajoympäristö sekä luokkakirjasto nimeltään Framework Class Library (FCL). Common Language Runtime on kaikkine ominaisuuksineen jokaisen tuetun ohjelmointikielen käytössä. CLR ei edes tiedä ajaessaan ohjelmaa, mitä kieltä on käytetty. Kääntäjälle on yhdentekevää mitä kieltä käytetään. Kääntäjä ainoastaan kiinnittää huomiota koodin oikeellisuuteen ja julkaisee näkemyksensä mukaisen toteutuksen siitä, mitä käyttäjä on komennoillaan tahtonut viestittää. Vaikka koodi muutetaankin yhteiseksi vä-

likieleksi (kutsutaan IL eli Intermediate Language Assembler), on syytä kuitenkin huomioida, että käyttäjän kannalta ei ole täysin sama asia tehdä tiettyä toteutusta millä tahansa kielellä, vaan mahdollisesti jokin tietty kieli toteuttaa kyseisen asian paremmin kuin toiset.(Richter 2003, xxi, 4.)

.NET Framework -luokkakirjasto eli .NET Framework Class Library (FCL) koostuu tuhansista tyyppimäärittelyistä, joista jokainen tarjoaa toimintoja. Yhdessä CLR ja FCL mahdollistavat useiden eri sovellustyyppien rakentamisen. Tyypit on luokiteltu niin, että samanaiheiset tyypit ovat saman nimiavaruuden alla (esimerkiksi System: Systemin alatyypit ovat erotettuina pisteellä ja ovat esimerkiksi tyyliä: System.Collections, System.IO tai System.Text). Näitä ovat: XML-pohjaiset Web Services -tyyppiset verkkopalvelut, Web Forms -sovellukset, joilla tehdään tavallisia verkkopalveluita, Windows Forms -sovellukset, jotka ovat Windows käyttöliittymäisiä sovelluksia, Windowsin konsolisovellukset (komentorivi- sovellukset), Windows palvelut, Itsenäisistä komponenteista koostuvat sovellukset, joissa voidaan käyttää edellä mainittuja sovellustyyppejä.(Richter 2003, 21–24.)

Edellä mainittujen tyyppien tarkoituksena on siis tarjota sovelluksille ja komponenteille toimintoja. Tyypit toimivat mekanismina, joka mahdollistaa eri ohjelmointikielillä toteutettujen koodiosien yhteisen kommunikaation. Tyypit ovat keskeinen osa CLR:ää ja niitä kutsutaan nimellä Common Type System (CTS). CTS myös antaa säännöt periytymiselle, olioiden elinkaarille ynnä muille mahdollisille toiminnoille. Edellä kirjoitettujen kappaleiden myötä sait varmasti perspektiiviä tulkita .NET-kehitystä myös pinnan alta. Itsessään kyseisten asioiden hallitseminen ei ole tämän työn kannalta tärkeää, mutta silti hyvä tietää, jotta ymmärtää .NET Frameworkin toimintaa ja osaa ymmärtää sen toimintaa verrattuna toisiin vaihtoehtoihin.(Richter 2003, 25–26.)

2.2 Suorituskyvyn parantamiseen liittyviä ohjeita

Lähes jokaisessa verkkosovelluksessa tulee vastaan tilanne, jossa halutaan säätää suorituskykyä. Kuitenkin ennen siirtymistä käytännön toimiin, on kartoitettava optimoinnin tarve ja ymmärrettävä ongelmakohdat. Suuret ja pienet toimet säästävät resursseja sananmukaisessa suhteessa, mutta vaikka suuret toimet saavat kerralla aikaan huomattavankin suuria hyötyjä, pienten toimien säästetyt millisekunnit ovat arvokkaita tilanteissa, joissa toiminto toteutetaan satoja tai tuhansia kertoja. Tarpeet on kuitenkin aina kartoitettava, sillä liiallinen optimointi voi johtaa resurssikulutuksen kasvuun tai olla hyödyltään minimaalinen kyseenomaista tarkoitusta varten.(Howard 2005)

Niin kuin aiemmin käsiteltiin, .NET-kehitystä voidaan toteuttaa monilla ohjelmointikielillä. Usein puhutaan yleisimpiin niistä, ja tämän projektin kannalta merkittävän, VB.NET-kielen (.NET’n Visual Basic) huonoudesta verrattuna muihin .NET-kehityksessä eniten käytettyihin ohjelmointikieliin. Esimerkkinä toinen VB.NET’n kanssa .NET-sovelluskehityksessä käytetty ohjelmointikieli C#, jota pidetään nopeamman koodin kannalta

selvästi parempana vaihtoehtona. Asiassa on toki perää, mutta erot eivät ole niin suuret kuin annetaan ymmärtää. Jos VB.NET-koodi rakennetaan oikeaoppiseksi, saadaan aikaan yhteneviä tuloksia kuin C#'lla. Samanlaiseksi kirjoitettu koodi tuottaa yhteneviä tuloksia.(Howard 2005)

2.2.1 ASP.NET Cache API ja välimuistinhallinta

Ensimmäisiä asioita, mitä on syytä tehdä, ennen kuin kirjoittaa riviäkään koodia, on tiedostaa välimuistin olemassaolo ja perehtyä sen hallintaan. Välimuistinhallinta ASP.NET-sovelluksissa toimii ASP.NET Cache API'n, eli rajapinnan, kautta. Välimuistinhallinta tulee kyseeseen tilanteissa, joissa dataa tarvitaan useammin kuin kerran. Välimuistiin siirtynyt data on nopeampaa palauttaa. Lisäksi välimuistilla hallitulla datalla on helpompaa kontrolloida datakuorman määriä, joita siirretään sovelluksen ja tietovaraston, kuten tietokannan, välillä. Lisäksi mikäli data ei muutu usein, on sen välimuistiin siirtämisestä enemmän hyötyä verraten dataan, joka muuttuu usein. Aina siis välimuistin hallintaan ei ole suotavaa turvautua. Välimuistinhallintaa voi myös käyttää liikaa. Tämä aiheuttaa ongelmia tilanteissa, joissa prosessoinnissa käytettävien resurssien ylärajat on tulossa vastaan. Suuri datakuorma tietovaraston ja sovelluksen välillä myös aiheuttaa kuormituspiikkejä. Vähintäänkin tämä hetkellisesti hidastaa sovelluksen toimintaa, mutta pahimmillaan tämä voi aiheuttaa järjestelmän kaatumispisteen ylittymiseen.(Howard 2005)

Suorituskyky on avainasemassa minkä tahansa sovelluksen luomisprosessissa. Parasta mahdollista suorituskykyä etsittäessä tärkeitä tekijöitä ovat esimerkiksi asiakaskoneen (client) ja palvelinpuolen kuorman keventäminen järkevällä välimuistinhallinnalla. Välimuistinhallinta on prosessi, jossa toistuvasti käytettyä dataa pidetään välimuistissa, jotta säästytään turhilta toistuvilta pyynnöiltä. Joskus voidaan myös säännöstellä pyyntöjen määrää. Palvelimelta välimuistiin otettua dataa on paljon nopeampaa käsitellä kuin dataa, joka tulee aina palvelimelta käsin. Välimuistin hallinta parantaa sovelluksen suorituskykyä, muuntautumiskykyä kasvaviin datamääriin sekä tiedon saatavuuteen. Järkevästi hienosäädetty sovellus toimii varmemmin ja paremmin kuin hienosäätämätön. ASP.NET tarjoaa mahdollisuuksia käsitellä välimuistia. On mahdollista käsitellä koko HTTP-vastausta käyttäen hallintatapaa nimeltä output caching. Muut vaihtoehdot ovat: partial page caching ja data caching. On kuitenkin myös syytä huomioida, että välimuistia voi hallita muutenkin kuin ASP.NET'n välityksellä. ASP.NET on sellaisenaan valmis hyödyntämään välimuistin hallintaa, joten mitään muutoksia ei ole pakko tehdä. Kuitenkin niin halutessaan sekin on mahdollista. Mikäli ei ole varma mitä on tekemässä, muutokset on parempi jättää tekemättä. Kuitenkin tarpeen vaatiessa tarvittavat tiedostot löytyvät konfiguraatiotiedostoista macine.config ja web.config.(Evjen, Hanselman & Rader 2008, 1039, 1049.)

Yksi tapa toteuttaa välimuistin hallintaa ASP.NET-kehityksessä on toteuttaa output caching. Tämän välimuistinhallintatekniikan tarkoitus on kyetä pitämään tietosisältöä muistissa tai levyllä myöhempää käyttöä varten. Output caching hallitsee kokonaista sivua. Tämä on hyödyllinen tapa hallita muistia, mikäli tietoa ei tarvita heti ja muutos esityssivulla voi tapahtua

viiveellä. Jos tapahtuman on syytä suoriutua kyseisellä sekunnilla tai muutamassa sekunnissa, ei ole suotavaa käyttää tätä tapaa, ainakaan ilman jatkokäsittelyä. Output caching voidaan ajastaa sekuntimääräisesti sopivalle aikavälille. Esimerkki: <%@ OutputCache Duration=”60”%>. Esimerkissä ajastus on tehty minuutin välein tapahtuvaksi (60 sekuntia). Tämä välimuistinhallintatapaa kontrolloidaan ASP-sivun kautta.(Evjen ym. 2008, 1039–1040.)

Partial page caching on periaatteeltaan vastaava kuin output caching, mutta suoritetaan vain tietylle osalle sivua. Se on hyödyllisempi tapa, mikäli sivun kautta hallitaan suurta datamäärää, joka tulee eri sisällöistä, ja koostetaan yhteen internet-sivulla. Tällöin on huomattavasti säästeliäämpää käsitellä vain ne osat sivua, jotka sitä tarvitsevat, kuin käsitellä kokonaista sivua. Toki on huomioitava, että tämän toimenpiteen hyödyt tulevat esiin vasta silloin, kuin käsiteltävää dataa on paljon ja tietolähteitä useampia tai dataa on vain paljon. Joskus output caching on parempi tapa.(Evjen ym. 2008, 1042–1043.)

ASP.NET sisältää objektin Cache. Sen kanssa, voidaan käsitellä välimuistissa säilytettäviä kohteita suoraan koodissa toteuttamalla niin sanottu data caching -tyyppinen välimuistin hallinta. Tämä objekti mahdollistaa datan muodostamisen edelleen monimuotoisemmiksi objekteiksi, kuten tietojoukoiksi ja kokonaisiksi ASP-sivuiksi. Esimerkkinä tiedon asettaminen välimuistiin seuraavaan tapaan käyttäen Visual Basic -kieltä: Cache(”VarastoitavaAsia”) = myDataSet. Myöhemmin muistiin siirrettyä kohdetta voidaan käyttää seuraavaan tapaan: Dim ds = new DataSet. Tässä luotiin uusi ”dimension” nimeltä ds. Seuraavaksi: ds = CType(Cache(”VarastoitavaAsia”), DataSet). Annetaan ds:lle arvo ja ohjeet eli tässä tapauksessa palautus muistista käyttöön. Tämä mahdollistaa välimuistinkäsittelyn manuaalisesti kehittäjän toimesta. (Evjen ym. 2008, 1048.)

2.2.2 Tilanhallinta

Aloitteleva ohjelmoija ei useinkaan vaivaa päätään ajatuksella tilanhallinnan tärkeydestä. Kuitenkin edetessään syvemmälle tulee huomanneeksi, että tilanhallinnallisiin asioihin ja niiden suunnitteluun kuluu tovi jos toinenkin. Tilanhallinta ei ole yksiselitteinen käsite, jossa on oikeita ja vääriä ratkaisuja, on vain uniikkeja tilanteita, joissa toinen ratkaisu on toista ratkaisua parempi. ASP.NET mahdollistaa kauttaan tilanhallinnan. On kuitenkin huomioitava, ettei se ole ainoa tapa hallita tilaa. (Evjen ym. 2008, 1003–1004.)

Tilanhallinnallisia vaihtoehtoja on useita. Osa on käytössä palvelimen puolella, osa clientin puolella. Jotta voi ymmärtää tilanhallintaa paremmin, on ymmärrettävä tilanhallinnan mahdollisuudet ja pyyntöjen elinkaari jokaisessa prosessin vaiheessa. Alussa selain tekee HTTP GET -pyynnön palvelimelle. Selain, joka mielletään tässä tapauksessa asiakkaaksi (client), ei ole käynyt sivulla aiemmin. Seuraavassa vaiheessa IIS (tai muu mahdollinen web-palvelin) ja ASP.NET-sovellus vastaavat pyyntöön tarjoamalla pyydettyä internet sivua asiakkaalle. Mahdollisesti mukaan lisätään eväste,

joka identifioi toiminnan asiakaskoneen ja palvelimen välillä. Tässä tapauksessa tila on asiakaskoneen ja palvelimen välinen pyyntöjä ja vastauksia sisältävä sopimus. Vastauksen HTML voi sisältää piilokenttiä. Ne toimivat samaan tapaan kuin evästeet, mutta ne lähetetään joka kerta uudelleen, kun sivu päivitetään tai muita toimia tehdään. Piilokentät ovat siis sivukohtaisia; evästeet toimialuekohtaisia. Evästeet palautetaan palvelimelle seuraavassa pyynnössä, HTTP POST. Tällöin kaikki kentät palautetaan, oli ne piilokenttiä tai ei. Evästeeseen laitettaan uniikki tunniste, jota voidaan käyttää avaimena palvelinpuolen tilanhallinnassa. Alla olevissa taulukoissa on esitetty tilanhallintakeinoja tarkemmin (Taulukko 1, Taulukko 2).(Evjen ym. 2008, 1004–1006.)

Taulukko 1. Palvelinpuolen tilanhallintavaihtoehdot.

PALVELINPUOLI	EDUT	HAITAT
Application State	Nopea. Jaetaan kaikkien käyttäjien kesken.	Tila varastoidaan kerran palvelinta kohti moniin palvelinten konfiguraatioihin.
Cache Object	Kuten Application State, mutta sisältää rajallisen ajan.	Tila varastoidaan kerran palvelinta kohti monissa eri palvelinten konfiguraatioissa.
Session State	Kolme vaihtoehtoa: in-process, out-of-process ja DB-backed. Voi olla myös evästeetön.	Voidaan käyttää väärin, evästeettömänä helpompi kaapata. Tarvitsee palvelinyhteyden ja prosessin päättämisestä tulee resurssikuluja.
Tietokanta	Päästään käsiksi jokaiselta puolelta palvelinkokonaisuutta.	Prosessin päättämisestä tulee resurssikuluja.

Taulukko 2. Tärkeimpien client- tilanhallintavaihtoehtojen edut ja haitat.

CLIENT	EDUT	HAITAT
Eväste	Helppokäyttöinen.	Selain voi kieltää käytön. Ei sovellu suureen datamäärään. Tiedon on oltava tietoa, joka saa vuotaa ulkopuoliselle. Jokainen HTTP-pyyntö ja HTTP-vastaus maksaa kokonsa verran resursseja.
Piilokenttä	Helppokäyttöinen. Soveltuu sivukohtaisen datan käsittelyyn.	Ei sovellu suureen datamäärään. Tiedon on oltava tietoa, mikä saa vuotaa ulkopuoliselle, eli ei sovellu herkälle datalle.
ViewState	Helppokäyttöinen. Soveltuu sivukohtaisen datan käsittelyyn.	Resurssitehokkuudesta huolimatta tällä on huono maine DataGrid-objektien kanssa.
ControlState	Helppokäyttöinen. Soveltuu sivukohtaisen ja tietyn kontrollin hallitsevan datan käsittelyyn.	Resurssitehokkuudesta huolimatta tällä on huono maine DataGrid-objektien kanssa. Vaatii ViewState'n kytkemisen päälle, vaikka muuten ei olisi tarvetta käyttää ViewStatea.
QueryString(URL)	Helpoin tapa ja usein tarpeen, jos sovelluksen käyttäjän on käytettävä URL-dataa suoraan selaimen välityksellä. (esimerkiksi esitettyjen tuotetietojen määrä sivulla voidaan säätää sopiaksi tällä tavoin).	Helppo loppukäyttäjän toimesta suoritettava hallinta voi olla toisissa tilanteissa ongelma. Ei voi sisällyttää suurta datamäärää, eikä sovellu herkälle datalle.

SessionState on resurssikuormituksen kannalta mahdollisuus, jonka kanssa on oltava huolellinen. Ensinäkin, jos sitä ei tarvitse, se on syytä ottaa pois käytöstä. Yksi tapa tehdä tämä on ASP.NET-sivulta käsin kirjoittamalla sinne komento <%@ Page EnableSessionState=”false” %>. Tapa estää SessionState'n sivukohtaisesti. Mikäli SessionState tarvitsee pääsyn session muuttujiin, muttei muokkaa niitä, false-arvon paikalle kirjoitetaa ReadOnly. Mikäli SessionState halutaan sulkea kokonaisesta ASP.NET-sovelluksesta, se tehdään web.config-tiedostossa kirjoittamalla: <sessionstate mode=”off”/>.(Developing High-Performance ASP.NET Applications n.d)

SessionState'sta on lisäksi hyvä tietää, että ASP.NET-sovelluksissa sitä voidaan käyttää kolmella tapaa. Valinnan kanssa on syytä olla huolellinen, sillä jokaisen niistä ominaisuudet ovat erilaiset. Ensimmäinen ja yleisin tapa on in-process (InProc), joka varastoituu HttpRuntime'n sisäiseen välimuistiin (120-bittinen merkkijono). Eli se kuormittaa muistia, mutta on luokiteltu nopeimmaksi tilanhallinnalliseksi tavaksi ASP.NET-kehityksen

kannalta. Mikäli prosessi syystä tai toisesta sulkeutuu, data katoaa. Mikäli syystä tai toisesta käytettävä tietokone vaihtuu, data ei välity käyttäjän mukana, vaan kyseessä on erittäin tapauskohtainen vaihtoehto. Tämä sessiotyyppi on suositeltava pienille sovelluksille, joissa sessiossa ei ole varastoituna tärkeää dataa. Mikäli halutaan käyttää toista sessiotyyppiä, niin kyseeseen tulee out-of-process. Niitä on kahta erilaista tyyppiä: Windows-palveluna jaettava versio tai SQL Serverin kanssa käytettävä versio (DB-backed tai SQL-Backed). Windows-palveluna käytettävä out-of-process hyödyntää aspnet_state.exe-tiedostoa, jolla palvelu aktivoituu. Kun out-of-process-sessioita hyödynnetään, web.config-tiedostossa ”SessionState mode” muutetaan arvoon ”StateServer”. SQL Serverillä toteutettu session tilanhallinta on kaikkein turvallisin, mutta hitain. Se soveltuu parhaiten suuriin palvelinympäristöihin. Kun tämä muoto on käytössä, ”SessionState mode” muutetaan arvoon ”SQLServer”. (Developing High-Performance ASP.NET Applications n.d; Evjen, ym. 2008 1008–1009, 1016–1017, 1022–1024 .)

2.2.3 Hae kerralla enemmän tietoa – säästä palvelinpyynnöissä

Tietokannassa on usein tietoa, jota haetaan useammin kuin kerran. Jokainen pyyntö tietokannalle nostaa suoritettavien hakujen määrää ja näin ollen resurssikuormitusta. Kuitenkin on mahdollista hakea yhdellä kertaa enemmän tietoa tietokannasta. Näin kommunikointiaika palvelimen ja sovelluksen välillä vähentyy. Haun mukana tulevaa tulosjoukkoa kutsutaan termillä tulosjoukko (result set). Yksi tehokas tapa on käyttää vuorovaikutteista SQL-tietokantakyselykieltä ASP.NET-sovelluksissa yhdessä Stored Procedures -tekniikan kanssa. Ne ovat SQL-komentosarjoja, joita voidaan liittää ASP.NET-sovellukseen. Stored Procedures’n kautta saatava resurssihyöty perustuu haettavaan datanmäärään ja ominaisuuksien hallintaan. Kun mahdollista, on järkevää käyttää SqlDataReader-luokkaa DataSet-luokan asemesta. Se vain lukee dataa, joten aina sen käyttö ei ole mahdollista, mutta on paljon tilanteita, jossa dataa vain luetaan, tällöin saadaan aikaan resurssisäästöä, kun hyödynnetään SqlDataReader-luokkaa.(Developing High-Performance ASP.NET Applications n.d; Howard 2005)

2.2.4 .NET, Stored Procedures, sivutettu data ja yhteydenhallinta

Kun halutaan hyödyntää resurssitehokkuutta ja käytössä on SQL Server, kannattaa käyttää Stored Procedures -tekniikkaa. Kun sovelluskehityksessä hyödynnetään .NET Frameworkia, suositellaan resurssitehokkaiden sovellusten luomiseen käytettäväksi SQL Server -tietokantasovellusta. Hyödyntämällä Stored Procedures –tekniikkaa, säästetään prosessoitavan koodin määrässä ja saavutetaan resurssitehokkuutta.(Developing High-Performance ASP.NET Applications n.d)

ASP.NET:ssa on ominaisuuksia, joilla on lisäominaisuus, datan sivutus. Se ei ole automaattisesti päällä, mutta se on helposti aktivoitavissa tarpeen tullen. Sivuttamalla dataa, saavutetaan paremmin käyttäjän tarpeita hyödyntävä tulosjoukko, jota näytetään vain tietty määrä kerrallaan. Lisä-

hyötynä navigointi helpottuu. Silti sivutuksessa on ongelmakohtansa: erittäin suuret palautetut datamäärät aiheuttavat ylimääräistä resurssikuormitusta, sillä kaikki data latautuu, vaikka sitä ei tarvitsekaan kokonaan. Sivutus yhdistettynä proseduureihin, kuitenkin auttaa tämän ongelman ratkaisussa ja saadaan samalla resurssitehokkuutta.(Howard 2005)

Web-sovellus tarvitsee yhteyden tietolähteeseen, jotta se voi toimia. TCP-yhteys sovelluksen ja tietokannan säilytyspaikan välillä on resursseja vaativa prosessi. TCP-yhteyden ei tarvitse olla automaattisesti aina päällä, vaan ainoastaan silloin kuin tehdään jokin toimitus sovelluksen ja tietokannan välillä. Esimerkiksi kyseessä voi olla tietokantahaku. Tästä syystä on kehitetty mahdollisuus jakaa yhteyksiä yhteisvarannosta tarpeen mukaan (Connection Pooling). Yhteys näin ollen käynnistetään useita kertoja session aikana, mutta ei ole auki turhaan. Näin säästetään tietoliikennekaistaa. On myös hyvä käyttää samaa yhteysmerkkijonoa yhteydenotossa. On suositeltavaa tehdä se käyttäjäperusteisesti. Yleisimmin tätä yhteyskäytäntöä suoritetaan suurissa verkkosovelluksissa.(Howard 2005)

3. RESURSSIKUORMITUSTESTAUKSEN TEORIA

Ennen käytäntöä, on syytä käsitellä resurssikuormitustestaus teoriassa. Seuraavaksi tarkastellaan sen kolmea käytössä olevaa testaustapaa. Ne ovat: kuormitustestaus, rasitustestaus ja kapasiteettitestaus. Prosessivaiheiden esittelyyn siirryttäessä perehdytään pienten sovellusten luonnissa käytettäviin vaihtoehtoihin tarkemmin.

3.1 ASP.NET-sovelluksen suorituskyvyn testaus

ASP.NET-sovellusten suorituskykyyn vaikuttavat monet tekijät. Kun sovellus otetaan käyttöön, on tiedettävä mahdollisimman tarkasti se, kuinka kyseinen sovellus käyttäytyy järjestelmässä, paljonko resursseja se vie ja kuinka kovaa kuormitusta se kestää. Tällöin tarvitaan testausta. Tässä luvussa tästä lähtien perehdytään testauksen saloihin suorituskykyyn perustuvan resurssikuormitustestauksen kautta. Myöhemmin tietoa sovelletaan selainratkaisujen kanssa opinnäytetyöhön liittyvässä käytännön projektissa.(Meier, Vasireddy, Babbar, & Mackman 2004)

Pääsääntöisesti suoriutuvuutta on testattava tilanteissa, jossa halutaan tietää ne rajat, joiden puitteissa sovellus toimii. Testauksella myös saadaan selville sen hetkinen kulutustilanne. Kuten jokainen sovellus, myös testaus on toteutettava yksilöllisesti tiettyä sovellusta ja käytettävissä olevaa laitteistoa mukaillen ja onkin suotavaa, että testausolosuhteet ovat lähellä todellista käyttötilannetta. Näin tiedetään, millaisia resurssipiikkejä järjestelmä kestää ja voidaan jopa ennakoida tulevaa: kuinka sovellus kykenee suoriutumaan muuttuneissa olosuhteissa (esimerkiksi kasvaneet käyttäjämäärät, tietokannat tai tietokantapyynnöt). Testauksen tehokkuus perustuu sitä edeltäviin suunnitelmiin ja testausskenaarioihin. Suorituskykytestausta voidaan tehdä kolmella tapaa: kuormitustestauksena (load testing), rasitustestauksena (stress testing) sekä kapasiteettitestauksena (capacity testing).(Meier ym. 2004)

Kuormitustestaus on tapa todentaa sovelluksen toimintaa vallitsevassa tilanteessa sekä tilanteissa, jossa kuormitukseen syntyy piikki. Näin voidaan todistaa sovelluksen kyky selviytyä siltä odotetuista tavoitteista. Tämä testauksen muoto kertoo täsmällisesti toteutettuna vastausaikojen (response time) aikakulutuksen, yleisen suoritusajan, resurssikuormitustason ja maksimaalisen toimintatason (ennen kuin järjestelmä kaatuu, jos kaataminen on ylipäätään mahdollista käytettävissä olevilla resursseilla). Joskus testaus voidaan myös soveltaa tilanteen mukaan vähäisemmäksi.(Meier ym. 2004)

Rasitustestaus arvioi rasitusmäärää normaali- ja äärimmäisoloissa. Näin paljastetaan mahdolliset järjestelmävirhekohdat, joita voi ilmetä, kun kuormitustasot ovat kriittisessä pisteessä. Kriittinen piste ei ole koskaan vakio, vaan riippuvainen järjestelmän tehokkuudesta. Rasitustestaus paljastaa järjestelmän heikot kohdat ja antaa valmiudet ymmärtää mahdollisia

virheitä tai antaa tietoa siitä mitä pitää korjata, jotta järjestelmä toimisi moitteettomasti tilanteessa kuin tilanteessa.(Meier ym. 2004)

Kapasiteettitestaus on kuormitustestausta täydentävä testausmuoto. Sillä saadaan selville palvelimen absoluuttinen kaatumispiste eli piste, jolloin palvelimen kuormitus on niin suuri, ettei se kykene jatkamaan toimintaansa, vaan sammuu hallitusti tai hallitsemattomasti riippuen varavirtajärjestelmästä tai sen puutteesta. Tämä auttaa kapasiteettisuunnitelmien luonnissa, joissa kartoitetaan tulevaisuuden kapasiteetti ja laitteistotarvetta (kuten suoritinteho, RAM-muistin määrä, levytilatarve ja verkkokaistan tarve). Kyseessä on erittäin vaativa testausmuoto, jota ei ole läheskään aina edes mahdollista toteuttaa. Tämänkin työn puitteissa tämä testausmuoto jää pois käytännön osiosta sekä suunnitelmista.(Meier ym. 2004)

3.2 Suorituskyvyn testaamisen tavoitteet ja testaamisen kohteet

Suorituskykytestauksen päätavoitteena on saada järjestelmä vastaamaan odotuksia niissä olosuhteissa, mitkä sille on luotu. On toki olemassa muitakin tavoitteita. Ensinäkin on syytä havaita mahdolliset pullonkaulat ja niistä aiheutuvat seuraukset. On myös tarpeen saada optimoitu ja hienosäädetty alusta ja toimintaympäristö, jossa on maksimoitu suoritusteho. Lisäksi on syytä todentaa sovelluksen luotettavuus rasituksen alla.(Meier ym. 2004)

Toimintakykyyn vaikuttavat itse sovelluksen lisäksi sen optimointi, sen alustan optimointi ja käyttövarmuus. Järjestelmä ei ole valmis, kun se on luotu toimivaksi, vaan sen syövereissä on vielä paljon säädettävää, ehkä jopa korjattavaa. Täydellisyyteen pyrkiminen on hyvä asia, mutta se on vaikeaa, ellei mahdotonta saavuttaa. Silti optimointitoimien väliin jättäminen ei ole järkevää. Aina toki sen väliin jättäminen ei kostaudu. Riskit ongelmiin ovat tällöin suuremmat.(Meier ym. 2004)

Täydelliseen testaukseen ei useinkaan riitä vain yksi suorituskykytesti, vaan testausta on kyettävä soveltamaan kunkin tilanteen vaatimusten mukaan. Improving .NET Application Performance and Scalability listaa asioita, joita testaamisen kautta voidaan havainnoida. Niitä ovat: vastausaika, suoriutumisteho ja maksimimäärä käyttäjiä, joita voidaan kerrallaan palvella. Lisäksi niihin kuuluvat järjestelmäresurssien tarve, se kuinka järjestelmä käyttäytyy kuormitustason muutoksissa sekä sovelluksen kaatumispisteen löytäminen. Myös liiallisesta rasituksesta johtuva hidastuminen tai sovelluksen toimintahäiriö, jonka oireet ja syyt tulevat osoitetuksi testauksen kautta kuuluvat näihin. Tarkoituksena on myös todentaa sovelluksen heikot kohdat eli niin sanotut pullonkaulat, sekä se minkälaista tukea tarvitaan, jotta järjestelmä kykenee toimimaan resurssikuormituksen kasvaessa.(Meier ym. 2004)

Suurin osa testauksesta on riippuvainen ennalta päätetyistä, dokumentoiduista ja sovituista kohteista, joihin testaus perustuu. Kun tietää mitä tekee ja mihin testaus vaikuttaa, saadaan koko testausprosessista enemmän irti. Kohteet on päätettävä tilannekohtaisesti, mutta Improving .NET Application Performance and Scalability esittelee muutaman peruskohteen,

jotka ovat yleisimmin mukana: vastausaika tai vastaanottoviive, suoriutumisteho, resurssien tarve (prosessoriteho, verkkokapasiteetti, levytila ja RAM-muisti) sekä työkuorma.(Meier ym. 2004)

3.3 Kuormitustestauksen prosessi ja kuusi testausvaihetta

Tässä testaustyypissä testataan kuormitusta normaaleissa ja kuormituspiikkiolosuhteissa. Lähtötasossa ollaan normaalikäyttötilassa ja kulutusta aletaan lisätä, kunnes saavutetaan huippukohta eli kynnys, jonka yli ei voida tai haluta mennä. Tuon kynnyksen ei ole pakko olla 90 – 100 % prosessoritehosta sillä on muistettava, että muutkin sovellukset tarvitsevat muistikapasiteettia. Testaus perustuu suunniteltuihin kuormitusmääriin, jotka päättyvät tiettyyn pisteeseen, joka on määritetty kynnyspisteeksi. Tätä kautta saataneen selvyys käyttökapasiteettirajoihin, joihin sovellus kykenee ilman, että sen ongelmakohdat katkaisevat toiminnan. Kuormitustestaus perustuu kuusivaiheiseen prosessiin. Ne ovat: avainskenaarioiden tunnistaminen, työkuorman identifiointi, mittauskohteiden määrittäminen, testaustapausten luonti, kuormitustilanteiden simulointi ja tulosten analysointi.(Meier ym. 2004)

Vaiheessa yksi kuormitustestaus aloitetaan tunnistamalla sovelluksen avainvaiheet eli avainskenaariot. Avainskenaarioihin luetaan ne alueet, jotka vaativat paljon resursseja tai ovat paljon käytettyjä. Näihin voi kuulua oikeastaan mitä tahansa, mutta esimerkkikohteita voivat olla esimerkiksi sisään kirjautuminen, tietojenhaku tietolähteestä, autenttisuuden tarkastaminen ja niin edelleen.(Meier ym. 2004)

Vaihe kaksi on tarvittavan resurssikuormituksen tunnistaminen eli työkuorman identifiointi, jotta voidaan määrittää myöhemmin työkuorma. Lopulta tiedoista kootaan työkuormamalli. Resurssikuormituksen määrä on syytä havaita ja tiedostaa, koska se on resurssitehokkuuden näkökulmasta kriittinen asia. Työkuormaan vaikuttavat muun muassa käyttäjien määrä, pyyntöjen määrä ja pyyntöjen kulkureitit. Kun alkukartoitukset on tehty, voidaan siirtyä testaamaan kuormituksen kestoa käyttämällä maksimaalista määrää käyttäjätilejä, joita sisään voi kirjautua, samanaikaisesti. Tämän jälkeen tuotetaan koko ajan lisää kuormaa, kunnes saavutetaan kynnyspiste eli piste, jossa havaitaan toimivuuden kärsivän huomattavasti.(Meier ym. 2004)

Työkuorman mallinnus on prosessi, jossa määritetyn työkuorman mukaan luodaan malli, johon kirjataan selvinneitä asioita ja tärkeitä kohtia. Jokaiseen työkuormamalliin liitetään vähintään seuraavat kohteet: avainskenaariot, yhtäaikaisten käyttäjien määrä järjestelmässä, pyyntöjen määrä ja pyyntöjen polut. Työkuormamalli määrittelee sen, kuinka jokainen määritelty skenaario toimii järjestelmässä. Esimerkki työkuormamallista löytyy tämän opinnäytetyön liitteenä (Liite 3).(Meier ym. 2004)

Vaihe kolme on mittauskohteiden määrittäminen. Mittauksia ei voi tehdä ilman mittausmääreitä. Jotta testauksessa on ylipäätään järkeä, on syytä verrata suoriutumiskykyä määreillä, jotka ovat mitattavissa. Ilman selvää rajaviivaa, ei testaustuloksia voida verrata keskenään. Testausta suunnitel-

lessa on siis tiedettävä valmiiksi mitä halutaan testata ja millä määreillä. On myös syytä päättää hyväksyttävät raja-arvot, jotka toimivat minimivaatimuksena tai keskivertosuoriutumisena. Kattava testausprosessi tutkii kaikki tärkeimmät testauksen kohteet: verkon testaus laitteistoineen, kohdetietokoneiden resurssit (prosessoriteho, RAM-muisti, kiintolevyn ja verkon toimivuus), alustaan liittyvät testit (esimerkiksi .NET Framework CLR ja ASP.NET:n toimintaa käsittelevät mittarit), sovelluksen toimintavarmuuteen liittyvät mittarit sekä sovelluksen kyky suoriutua asetetuista tavoitteista. Kun tarvittavan datan saamiseksi on löydetty tarvittava määrä soveltuvia mittareita, siirrytään eteenpäin. Tällöin määritetään testaukseen liittyvien normaali ja piikkiolosuhteiden rajat.(Meier ym. 2004)

Vaihe neljä on testaustapausten luonti. Tämä vaihe liittyy olennaisesti vaiheeseen kaksi. Kun toteutettavat testaustapauksen on määritetty, ne on syytä dokumentoida. Dokumentaatio voi sisältää mitä tahansa tietoa, mutta se on syytä pitää selkeänä. Dokumentaation on ainakin syytä sisältää seuraavat asiat: testattava kohde, käyttäjämäärä, testausaika ja sovelluksen suoriutumisaika kyseisestä tehtävästä.(Meier ym. 2004)

Vaihe viisi on kuormitustilanteiden simulointi. Tässä vaiheessa käytetään erillistä työkalua, jolla saadaan mitattua kuormitustasoa. On syytä huomioida, ettei testauskoneiden kuormitus ole jo valmiiksi lähellä kriittistä pistettä. Sillä voi olla häiritsevä vaikutus tulokseen. Paras tulos saavutetaan puhtaalla alustalla, jossa ei ole häiriötekijöitä, ellei testauksessa haluta tutkia kyseisten häiriötekijöiden vaikutusta prosessiin.(Meier ym. 2004)

Vaihe kuusi on tulosten analysointi, jossa arvioidaan saadut tulokset ja tehdään tarvittavat johtopäätökset. Tehdyt testit ja niiden tulokset mahdollistavat vallitsevan tilanteen analysoinnin lisäksi mahdollisuuden arvioida tulevaisuutta. Tällöin on toki tehtävä oletuslaskelmia. Tämän lisäksi testausten kautta voidaan paljastaa sovelluksen heikot kohdat. Hyvän testauksen kautta saadaan aikaan parempia lopputuloksia.(Meier ym. 2004)

3.4 Rasitustestauksen prosessi ja kuusi testausvaihetta

Rasitustestaus perustuu sovelluksen asettamiseen suuren rasituksen alle. Näin tehdään, jotta löydetään mahdolliset kriittiset virhekohteet, jotka voivat kaataa järjestelmän. Rasitustestauksen rakenteet ovat samanlaiset kuin edellä esitellyssä kuormitustestauksessa. Alla esitellään rasitustestauksen kuusi kohtaa.(Meier ym. 2004)

Vaiheessa yksi rasitustestaus aloitetaan tunnistamalla sovelluksen avainvaiheet eli skenaariot. Mukaan otetaan kaikki ne skenaariot jotka pitää testata. Skenaariot tulee valita sen mukaan, kuinka kriittisiä ne ovat sovelluksen toimintakyvylle. Toisekseen niihin on suotavaa valita toiminnot, jotka joutuvat todennäköisimmin suorituskyvyn riittävyyden armoille. Esimerkiksi näihin voivat kuulua tietokantahaut. Skenaariot pitää myös valita perustuen kuormitustestauksen mukaan pisteisiin, joissa on todennäköisimmin ongelmakohtia. Testaus on syytä tehdä vielä suorittamasi optimointihienosäädön jälkeen, jotta näet aikaan saadut tulokset.(Meier ym. 2004)

Vaihe kaksi on tarvittavan työkuorman tunnistaminen, kuten myös osana kuormitustestausta. Rasituskuormituksen määrä on syytä havaita ja tiedostaa. Rasitustasoa on syytä kasvattaa haluttuun pisteeseen. Jopa ylittää kynnyspiste. Jokainen prosessi, joka on käynnissä, vaatii oman osansa kokonaisresurssimäärästä. Useamman prosessin yhtäaikainen pyörittäminen auttaa kriittisen pisteen saavuttamisessa.(Meier ym. 2004)

Vaihe kolme on mittausmääreiden määrittäminen. Mittauksia ei voi tehdä ilman mittausmääreitä. Jotta testauksessa on ylipäätään järkeä, on syytä verrata kykyä määreillä, jotka ovat mitattavissa. Ilman selvää rajaviivaa, ei testaustuloksia voida verrata keskenään. Rasitustestaus perustuu suurin osin RAM-muistin käyttöön ja käytettyyn muistivirtaan. Ne saadaan selville käyttämällä aiheeseen sopivaa mittausvälinettä.(Meier ym. 2004)

Vaihe neljä on testaustapausten luominen. Tämä vaihe liittyy olennaisesti vaiheeseen kaksi. Kun toteutettavat testaustapauksen on määritetty, ne on syytä dokumentoida. Dokumentaatio voi sisältää mitä tahansa tietoa, mutta se on syytä pitää selkeänä. Dokumentaation on ainakin syytä sisältää seuraavat asiat: testattava kohde, käyttäjämäärä, testausaika ja sovelluksen suoriutumisaika kyseisestä tehtävästä.(Meier ym. 2004)

Vaiheessa viisi kuormitustilanteet tulee simuloida. Tämä tarkoittaa testaajan luomaa analyysia testaustilanteista. Tässä vaiheessa voidaan käyttää erillistä työkalua, jolla saadaan mitattua kuormitustasoa muuttuneissa tilanteissa. On syytä huomioida, ettei testauskoneiden kuormitus ole jo valmiiksi lähellä kriittistä pistettä.(Meier ym. 2004)

Lopuksi, vaiheessa kuusi, arvioidaan saadut tulokset ja tehdä tarvittavat johtopäätökset. Tehdyt testit ja niiden tulokset mahdollistavat vallitsevan tilanteen analysoinnin lisäksi mahdollisuuden arvioida tulevaisuutta. Tällöin on toki tehtävä oletuslaskelmia. Kun testaus paljastaa kriittisiä ongelmakohtia, on niihin löydettävä ratkaisu, jolla ne saadaan eliminoitua, tai edes minimoitua, niiden mahdollinen toteutuminen.(Meier ym. 2004)

4. TESTAUKSEN SUUNNITTELU

Tässä luvussa perehdytään kahteen testauskokonaisuuteen täsmällisemmin. Ne ovat kuormitustestaus ja rasitustestaus. Kolmas mahdollinen tapa, kapasiteetti testaus, jätetään pois. Se ei ole resurssien kannalta tässä tapauksessa mahdollista, eikä perustellusti järkevääkään, sillä toteutetut sovellukset ovat suhteellisen pieniä ja kevyitä. Seuraavaksi syvennytään seikkoihin, joita on käsiteltävä ennen todellista testaustilannetta.

4.1 Kuormitustestaus

Aiemmin tässä opinnäytetyössä käsiteltiin kuusivaiheinen teoria, jonka tavoitteena oli avata prosessin teoria. Kuitenkin itse testausprosessi on yksityiskohtaisempi. Käytännöntestausta tehdessä on syytä huomioida, että testausta voidaan suorittaa monella tapaa ja monella tarkkuudella. Yleisin tapa on testata järjestelmää normaaleissa ja korkean rasitusluokan olosuhteissa, jopa rasitushuipussa. Testauksen alkuvaiheessa on syytä aloittaa pienestä käyttäjämäärästä ja kasvattaa käyttäjämäärää edeten kohti huippukuormitusolosuhteita. Kun testi jatkuu tarpeeksi pitkälle, kynnyspiste lopulta ylittyy. Kuormaraja voidaan määrittää prosentuaalisesti koskemaan jotain tiettyä prosentuaalista määrää maksimaalisesta palvelinkuormasta tai ottaa mittausmääreeksi jokin sekuntimäärä, joka pyyntöön vastaamiseen kuluu. Esimerkiksi saadaanko haku valmiiksi alle kolmessa sekunnissa. Todelliseen testausprosessiin sisältyy vaiheita, jotka eivät olleet esillä silloin kuin asiaa käytiin läpi teoriassa. Kuormitustestauksen päätarkoituksena on löytää sovelluksen maksimaalinen toimintakapasiteetti. Tämän testausprosessin käytännönosaan valmistavat vaiheet ovat: hyväksyttävien kriteerien kartoitus (1), avainskenaarioiden tunnistaminen (2), kuormitusmallin tekeminen (3), kohdekuormitustasojen määrittäminen (4), mittausmääreiden määrittäminen (5), yksilöllisen testauksen suunnitteleminen (6), testauksen suorittamien (7) sekä tulosten analysointi (8). (Meier, Farre, Bansode, Barber & Rea. 2007.)

Hyväksyttävissä rajoissa olevien kriteerien kartoittaminen on tärkeimmillään sovelluksen kehityksen alkuvaiheessa. On hyvin tärkeää, että nämä arvot on dokumentoitu. Tämän vaiheen testauskohteita ovat: suoriutumiselle asetettu aikaraja, sekuntia kohden käsiteltävät toiminnot, resurssikuormituksen ja kulutuksen määrittäminen sekä se kuinka monta käyttäjää voi olla kirjautuneena sisään ja suorittaa toimiaan kyseenomaisella laitteistolla.(Meier ym. 2007.)

Skenaarioilla tarkoitetaan käyttäjän suorittamia arvioituja käyttäjäpolkuja, joita hän suorittaa yhden käyttösession aikana. Tietyt vaiheet ovat sovelluksen toiminnalle kriittisimpiä. Kriittisimpiä ovat ne polut, joihin liittyy suuri resurssikuormitus. Niitä kutsutaan avainskenaarioiksi. Avainskenaarioihin liittyy suoritusvarmuuteen liittyviä vaateita ja niiden toiminnan lakkaaminen aiheuttaa kriittisiä toimivuusongelmia.(Meier ym. 2007.)

Käyttötapauksista määritetään navigaatiopolku, joka sisältää perustoimet, joita käyttäjä tekee tietyn tapauksen toteuttamiseen. Tämä voidaan mieltää myös yhden session aikana toteutuneiksi toimiksi, mikäli toimet sisältyvät enemmän kokonaiseen sessioon kuin yksittäisiin toimiin. Esimerkiksi tämän opinnäytetyön selainsovellusten kanssa tällainen sessiokohtaisuus on parempi tapa, sillä yksittäisiä toimia ei ole paljon. Suuremmissa sovelluksissa yksityiskohtaisuus on järkevämpi tapa toimia. Sessioilla on kestonsa. On syytä määrittää oletussessiolle keskiaika, tai jopa ajastaa sessio tietyn kestoiseksi. Näissä tapauksissa on myös ajateltava käyttäjää: tietyissä osissa käyttäjä tarvitsee miettimisaikaa tai aikaa muuten vain kuluu enemmän sovelluksen tietyissä kohdissa verrattuna toisiin. Käyttäjän sovellustuntemus ja taitotaso ovat tekijöitä, joilla on huomattavaa vaikutusta suoriutumisaikaan. On myös huomioitava, että olemassa eritasoisia käyttäjiä, jotka toteuttavat erilaisia prosesseja.(Meier ym. 2007.)

Edellisessä kohdassa käsitellyn kuormitusmallin kautta edetään vaiheeseen, jossa vedetään kuormitusmallin asiat yhteen. Niiden kautta on syytä tehdä tarvittavat johtopäätökset kuormitukseen vaikuttavista tekijöistä. Tällä toimella halutaan saavuttaa tietoisuus siitä, miten eritasoisin toimin voidaan saada aikaan erilaisia kuormitustasoja. Näin myöhemmin toteutettavista testeistä saadaan enemmän irti ja voidaan verrata eri toimijoiden toimia toisiinsa paremmalla lopputuloksella.(Meier ym. 2007.)

Mittausmääreitä on paljon. On siis hyvä tietää ennakkoon kuinka tarkkoja tuloksia haluaa ja toisaalta katsoa, että mittausmääreet ovat vertailukelpoiset, jotta vertailua voidaan toteuttaa. Mittauksia ei voi tehdä ilman mittausmääreitä. Ilman selvää rajaviivaa, ei testaustuloksia voida verrata keskenään. Rasitustestaus perustuu suurin osin RAM-muistin käyttöön ja käytettyyn muistivirtaan. Ne saadaan selville käyttämällä aiheeseen sopivaa mittausvälinettä.(Meier ym. 2007.)

Edellä mainittujen kohtien kautta lopulta suunnitellaan testaus. Testaus tulee yksilöidä tapauskohtaisesti tapausten, määreiden ja kuormitusanalyysin kautta. Tulee huomioida, että erilaisilla testeillä pyritään erilaisiin tuloksiin, sillä niillä on omat tarkoituksensa sekä niiden tarkoituksena on saada erilaista tietoa kohteestaan. Testin suunnittelun on onnistuttava niin, että testistä saadaan irti halutut määreet, jotka on helppoa ymmärtää, joita on helppoa analysoida ja joiden pohjalta voidaan tehdä jatkokehitystä sovellukselle. Lisäksi on kiinnitettävä huomiota seuraaviin seikkoihin. Testaussuunnitelmaan tulee sisältää kaiken sen tiedon, mitä aidon testin toteuttamiseen vaaditaan. On myös harkittava sellaisen tiedon lisäämistä testausprosessiin, joka ei liity testaustoimenpiteen välttämättömiin tapahtumiin. Esimerkiksi kaikki käyttäjät eivät kirjoita salasanaansa oikein heti ensimmäisellä kerralla, ja toiset saattavat tehdä virheellisiä toimia järjestelmässä, ennen oikein suoritettua tapahtumaa. Lisäksi uudet käyttäjät kuluttavat aikaa enemmän kuin vanhan rutinoituneet käyttäjät. Muista myös ajatella muutakin kuin loppukäyttäjää. Vaikka sovelluksen käyttäjäystävällisyys on hyvä asia, on myös syytä ottaa huomioon järjestelmän tekemät prosessit. Nämä prosessit (kuten päivitykset tai muut toimet) lisäävät kuormitusmäärää, mikäli sisällä on samaan aikaan käyttäjiä. Myöskään liialliseen täydellisyyteen tai tiedon liialliseen yksinkertaistamiseen ei tule sortua tes-

tausta toteuttaessa. Testiä voi kehittää myös testauksen aikana tulosten myötä, jos tarve niin vaatii.(Meier ym. 2007.)

Lopulta voidaan suorittaa itse testaus. Testauksen kannalta on parasta olla mahdollisimman aidon oloisessa (ei pakosti aidossa) ympäristössä. Mikäli eroavaisuuksia on, ne on syytä dokumentoida analysointituloksiin. On myös syytä huomioida, että testausprosessissa kuormitusmuutokset on kannattavaa tehdä pienin harppauksin. Mitä suurempi järjestelmä testattavana on, sitä tarkempaa on varmistua asianmukaisesta menettelystä.(Meier ym. 2007.)

Tulosten kautta järjestelmän heikot kohdat ilmenevät ja niihin voidaan vaikuttaa. Toteutetut testit ja niiden tulokset mahdollistavat vallitsevan tilanteen analysoinnin lisäksi mahdollisuuden arvioida tulevaisuutta, mikäli tehdään oletuslaskelmia. Kun testaus paljastaa kriittisiä ongelmakohtia, on niihin löydettävä ratkaisu, jolla ne saadaan eliminoitua tai mahdollinen toteutuminen pidetään kontrollissa.(Meier ym. 2007.)

4.2 Rasitustestaus

Seuraavaksi käydään läpi ne toimet, joita on otettava huomioon suoritettaessa käytännön testausta. Osa on jo käyty läpi aiemmissa teoriaosuuksissa, mutta kuten edellä, tässäkin tapauksessa on syytä perehtyä täsmällisemmin aiheeseen ennen todellista testaustilannetta.(Meier ym. 2007.)

Rasitustestauksen päätarkoitus on todentaa verkkosovelluksen fyysinen resurssikuormitustarve, saatavuus ja luotettavuus suurta kuormitusta aiheuttavissa tilanteessa. Tähän voivat vaikuttaa latausmäärän suuruus, käyttäjämäärät sekä järjestelmän alhainen toimintakapasiteetti. Rasitustestauksen tarkoitus onkin nähdä sovelluksen toiminta silloin, kun se on rasituksen alainen. Kunnolla toteutettu rasitustesti paljastaa muun muassa seuraavia vikoja: tahdistuksen ja ajastuksen toimintahäiriöt, häiriöpisteiden löytäminen, prioriteetteihin liittyvät ongelmat ja toimintahäiriöt jotka johtavat tietovuotoihin (tilanteisiin joissa data ei mene haluttuun paikkaan vaan katoaa matkalla). Tämän testausprosessin tarkoituksena on siis löytää ne toimintaa haittaavat virhekohdat, jotka voivat kehittyä haitaksi myöhemmin, kun kuormitus on kasvanut. Seuraavaksi mainittakoon muutamia esimerkkejä ongelmista, joita voi ilmetä kovan rasituksen aikaisissa tilanteissa: Erittäin suuret kuormitusmäärät datan tai käyttäjien toimesta. Näitä ilmenee muun muassa palvelunestohyökkäyksissä tai muissa tilanteissa, missä tiettyyn aikaväliin tulee äärimmäisen suuria määriä dataa aiheuttamaan tietotulvaa, joka lopulta estää toiminnan täysin. Lisäksi yllättävä resurssikapasiteetin lasku laiterikon tai muun syyn seurauksena voi aiheuttaa näitä ongelmia. Tällaiset tilanteet johtavat lopulta ikäviin seurauksiin, kuten datan virheisiin tai suoranaiseen katoamiseen, rasituksen loputtua tulee pyyntötulva, joka haittaa ensimmäisiä hetkiä rasituksen tasaannuttua, sovelluksen osat eivät vastaa pyyntöihin, käyttäjä saa eteensä joukon virheilmoituksia asioista, joita sovellus ei ole kyennyt toteuttamaan.(Meier ym. 2007.)

Todelliseen testausprosessiin sisältyy uusia vaiheita, jotka eivät olleet esillä silloin kuin asiaa käytiin läpi teoriassa. Prosessi toimitetaan seuraavin kohdin: testauskohteiden identifiointi (1), avainskenaarioiden tunnistaminen (2), kuormitusmäärä identifiointi (3), mittausmääreiden määrittäminen (3), yksilöllisen testauksen suunnitteleminen (4), testaustapausten luominen (5), tarvittavan kuormituksen simulointi (6) ja tulosten analysointi (7).(Meier ym. 2007.)

Aluksi lähdetään liikkeelle testauksen kohteista ja niiden määrittämisestä. Kohteiden tavoitteiden määrittelyssä auttaa seuraavat kysymykset. Onko kyseisen testaustoimen tarkoituksena todentaa ne asiat, jotka voivat pettää pahoin seurauksin? Pitääkö testin tuottaa tietoa, jonka avulla voidaan välttää katastrofaalisia virheitä? Pitääkö testin todentaa se, kuinka sovellus toimii, kun muisti, levytila, verkkokaista tai prosessori toimii vajaalla tasolla? Pitääkö testin todentaa se, ettei testattava toiminto petä rasituksen alla? Näiden kysymysten kautta, on mahdollisuus saada haluttuja identifioitua tärkeimmät kohteet. Testausta on mahdollista toteuttaa monella tavalla ja jokaisessa tilanteessa prioriteetit ovat ennalta määrittelemättömät.(Meier ym. 2007.)

Avainskenaarioiden tunnistamisen tarkoituksena on löytää kohteet, jotka tarvitsevat erityishuomiota. Näitä ovat kohteet jotka ovat kriittisiä järjestelmän suorituskyvyn kannalta, ovat kohteita, jotka ovat suorituskyvyn armoilla (eli vaativat paljon tehoa) ja ovat kohteita, joihin suurella todennäköisyydellä liittyy toiminnallisia riskejä. Tällaisia voivat olla esimerkiksi sisään kirjautuminen, asiakastietojen hakeminen tietokannasta tai muusta tietolähteestä sekä tiedon lisäys, muokkaus tai poisto. Kun kohteet on selvitetty, voidaan luoda työkuormaprofiilit.(Meier ym. 2007.)

Kussakin sovelluksen kautta suoritettavassa toimessa on oma resurssikuormituksensa. Jossain tapauksissa voidaan mennä sovelluksen sietokyvyn rajoille. Rajat saadaan selville pienin harppauksin etenevin rasituslisäyksin. Tärkeitä on, että testaus suoritetaan muuttuvin rasitusmäärin, kunnes päästään virhekohtaan. Tähän voidaan päästä lisäämällä käyttäjiä, kasvattamalla tehtyjen toimien rasittavuutta sekä kasvattamalla toimintojen määrää. Testausaika voi vaihdella tarpeen mukaan. Esimerkkinä artikkeli tarjoaa toistuvat sisään kirjautumiset 30 sekunnin aikana. Testausajan ei tarvitse kestää tunteja tai päiviä. Lyhyestäkin testistä saa tuloksia tulevaisuuteen, mikäli niiden lisäksi suoritetaan laskelmia. Identifiointi prosessia helpottamaan esitän seuraavaksi kohtia artikkelista. Niitä ovat tiedosta sisään kirjautuvien eritasoisten käyttäjien lukumäärät, sillä tehdyillä toimilla on merkitystä lopputuloksen kannalta. On myös hyväksi arvioida kuormituksen huipputasot: arvioi tai laske prosentuaalinen kuormitusmäärä kussakin avaintilanteessa ja identifioi tilanne, missä saavutetaan maksimitaso, jolla sovellusta voidaan kuormittaa. Määritä myös normaalin kuormituksen pohjalta se kuormamäärä, mikä muihin toimiin on tarjolla. Näin testausta ei tarvitse suorittaa täydellä painolla, vaan tulokset voidaan arvioida, kun tiedetään kuormituksen perustasot kussakin tilanteessa. Tämä tapa toimii hyvänä aloitustasona rasitustestauksella ja antaa kuvan siitä, mitkä on sen hetkinen kuormitustilanne ja paljonko on varaa lisätä

kuormitusta lähiaikoina. Täsmällistä kaatumispistettä ei toki saada selville. (Meier ym. 2007.)

Mittausmääreet on syytä valita tarkasti. Määreet on syytä päättää tilannekohtaisesti, mutta rasitustestauksen näkökulmasta testausta on suoritettava seuraavilla tavoilla seuraavanlaisiin kohtiin. Prosessin läpivienti: muistin kulutuksen määrä, prosessorin kuormitus ja muistin vapautuminen muuhun käyttöön. Muistin hallinta: käytettävän muistin määrä ja muistin käyttö. Ota myös huomioon levytilan suuruus sekä käytettävän verkkokaistan suuruus sekä toiminnoista suoriutuminen tai suoriutumatta jääminen kussakin tilanteessa. Myös vastausajat ovat tärkeitä.(Meier ym. 2007.)

Seuraavaksi suunnitellaan testit ja suoritetaan ne. Testaustietoja voidaan merkitä numeerisestikin, mutta tietyissä tapauksissa riittää tieto siitä, että sovellus suoriutuu tehtävästä tai jättää suoriutumatta siitä. Testit ovat yleensä tapauskohtaisia ja testit ovat paljolti riippuvaisia testauksen luonteesta ja siitä mihin sillä halutaan pyrkiä. Aina ei tarvitse luoda suurta määrää analyysidataa, jotta saadaan tarvittava tieto siitä, mitä pitää tehdä.(Meier ym. 2007.)

Tulosten pohjalta voidaan suorittaa arvioita tulevaisuuden varalle. On hyvä muistaa seuraavat vaiheet: valitse mahdollisimman todenmukainen ympäristö testaukseen, varmista, että testausprosessi ja testausympäristö soveltuvat testaustilanteeseen., tarkista kaiken toimivuus, ellei ole tarkoitus, muista käynnistää järjestelmä uudestaan juuri ennen testausprosessin alkua. Lopuksi analysoidaan tulokset. Saatujen tulosten kautta analysoidaan suoriutumista vaadituista tehtävistä ja katsotaan onnistuiko kaikki halutulla tavalla. Saatujen tulosten kautta voidaan korjata ongelmakohtia ja pitää näin huoli, ettei sama toistu käytössä.(Meier ym. 2007.)

4.3 Rasitustestauksen erilaisia muotoja

Rasitustestausta voidaan suorittaa monella tavalla. Seuraavaksi käydään läpi yleisesti tunnettuja tapoja suorittaa rasitustestaus. Yksi tapa on suorittaa sovelluskohtainen rasitustestaus. Kokonaisen sovelluksen rasitustestaaminen, ei käsittele mitään prosessia yksilöllisesti, vaan kattaa yhdessä testausprosessissa kokonaisen sovelluksen. Tällä tavoin saadaan selville toimivuuden rajat ja tilanne, jossa sovellus ei toimi, mutta tarkka syy jää selvittämättä. Jos halutaan tutkia tarkkoja syitä, tämä ei ole hyvä tapa testata, mutta tätä tyyliä käytetään paljon suuren kuormitustestausprosessin täydentämiseen tai kapasiteetin suunnittelun osana.(Meier ym. 2007.)

Toiminnallisuuksien rasitustestaus keskitetään sovelluksen tärkeimpien toiminnallisuuksien rasituksen alle asettamiseen. Tarkoituksena on eristää ongelmat, jolloin todellinen ongelman syy saadaan selville, mutta tämä on aikaa vievä prosessi samoin kuin sovelluspohjainen testaus. Tämä on hyvä tapa viimeistellä optimointia ja hienosäätää järjestelmää.(Meier ym. 2007.)

Järjestelmän osien sisäisessä rasitustestauksessa mukaan otetaan monia tai jopa kaikki järjestelmän osat (ei vain yksi sovellus) ja pyritään saamaan aikaan suuren luokan kuormitustasoja. Tällä saadaan selville järjestelmän

sietokyky sekä se, haittaavatko eri järjestelmän osat toistensa toimintaa. Kokeellinen rasitustestaus on lähestymistapa rasitustestaukseen, jossa pyritään aiheuttamaan järjestelmälle, sovellukselle tai tietylle komponentille suurempi rasitus, kuin on todennäköistä saavuttaa, mutta ei ole periaatteessa täysin mahdotonta.(Meier ym. 2007.)

5. SELAINRATKAISUT, INTEGRAATIO, TESTAUS JA TULOKSET

Hämeen ammattikorkeakoulun toimeksiannon pohjalta syntyi projekti, jonka yhtenä lopputuotteena syntyi kaksi kappaletta selainratkaisuja. Toimeksiannon tarkoituksena oli kartoittaa niiden resurssitehokkuutta vertaillen sovelluksia toisiinsa. Jotta vertailu voitiin toteuttaa, tuli luoda testaussuunnitelma ja sen pohjalta toteuttaa tarvittava testaus. Selvitykseen piti liittää vielä ratkaisu siitä, kuinka Microsoft Dynamics NAV -integraatio kannattaa toteuttaa Internet Explorer -selaimella käytettävälle, .NET- teknologian avulla toteutetulle, internet-sivulle.

Selainratkaisuiden toiminnallisuuksien tuli toteuttaa asiakkaiden lisäyksen, päivityksen ja poiston tarvitsemat toimet. Lisäksi sovelluksesta tuli löytyä raportointiominaisuus, jonka avulla asiakkaiden lukumäärä paikkakunnittain voitiin lukea sekä graafisessa muodossa että taulukkomaisessa muodossa.

5.1 Selainratkaisujen suunnittelu

Toimeksiannossa tekijää ei sidottu täsmällisesti noudattamaan mitään tarkkaa linjaa, joten projektin läpiviemiseen oli vapaat kädet. Näin ollen tehtävänkuvaan kuului yhtenä aspektina vapaus päättää selainratkaisujen rakenteesta ja käytetyistä tekniikoista, kunhan toimeksiannon ehdot täyttyvät. Työllä oli kuitenkin tiettyjä raja-arvoja. Kun vaatimusmäärittelyn kaltaisia vaihtoehtoja alettiin kartoittaa, keskeisiksi seikoiksi nousivat tehtävänannon vaatimukset sekä selainratkaisuiden keskinäinen vertailtavuus. Lisäksi ratkaisut oli luotava niin, että niiden kautta saatiin dataa resurssitehokkuuteen liittyen. Ratkaisuiden tuli olla resurssitehokkaita, että toisiinsa nähden sellaista, että niistä saatiin eriteltyä vertailukelpoista dataa.

Luontiprosessin alkuvaiheessa määrittyi myös se, että projekti pitäytyi Microsoftin tekniikoissa selainratkaisuiden toimivuuden maksimoimiseksi. Microsoftin tuotteilla on taipumusta toimia parhaiten toisten Microsoftin tuotteiden kanssa, usein muut tuotteet häviävät tavalla tai toisella toimintavarmuudessa ja resursseissa Microsoftin vastaaville yhteisprojekteissa. Tämä kaikki liittyi myös olennaisena osana pohdintaan tulevasta selainratkaisuiden integraatiota NAV’n kanssa. Lisäksi alkukartoitukset paljastivat, että tuleva integraation kautta saatava data piti kaivaa esiin SQl Server -tietokannasta.

Tuloksena syntyi kaksi samantyyppistä, keskenään verrattavissa olevaa, mutta kuitenkin erilaista ratkaisuvaihtoehtoa. Käyttämättä jäi monia vaihtoehtoja, joissa oli yhtälailla potentiaalia ja perustetta tulla valituksi kyseisen projektiin vaihtoehtokandidaatiksi. On selvää, että kyseiset ratkaisut eivät ole kaksi absoluuttisesti parasta vaihtoehtoa, vaan kaksi vaihtoehtoa, jotka on valittu vertailtavuuden ja todennäköisen toimeksiannonmukaisen suoriutumisen näkökulmasta. Tämän projektin laajuus ei kattaisi täydellis-

tä kartoitusta. Silloin selainratkaisuvaihtoehtoja tulisi luoda kymmeniä, ellei satoja ja niiden testausprosessi ottaisi aikansa.

Tähän projektiin valitut tekniikat ja konseptit on valittu asiantuntevan taustamateriaalin ja valinnanvapauden määrittämänä. Kahteen vaihtoehtoon ei saa mahtumaan kaikkea tarjolla olevaa, joten paljon rajautui tästä syystä pois. Pidin kuitenkin optimaalisena rajata projektin kahteen sovellukseen ajankäytöllisistä ja materiaalin laajuuteen liittyvistä syistä. Kolmannen ratkaisun mukaan ottaminen ei olisi ollut testauksen näkökulmasta katsottuna edes mahdollista, lisäksi otantana kaksi tai kolme sovellusvaihtoehtoa on suhteutettuna kymmeniin vaihtoehtoihin suhteellisen näennäinen ero. Hyödyn näkökulmasta katsoen se ei myöskään maksa vaivaa.

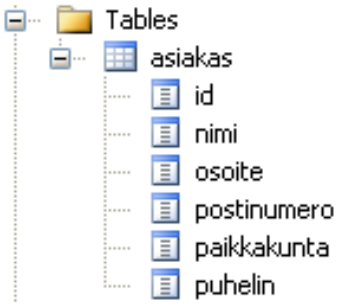
Projektissa ASP.NET-versioksi valittiin ASP.NET 3.5. ASP.NET 3.5 oli projektin alkaessa uusin ASP.NET-versio. Tilanteessa oli oletettavaa, että versio 2.0 tulee jäämään vanhaksi, joten sitä ei ollut enää kannattavaa sisällyttää tähän projektiin. Lisäksi oli todennäköistä, että kehitysevoluutio tuo uuden version markkinoille tulevaisuudessa. Projektia päättyessä markkinoille oli tullut jo versio 4.0. .NET 3.5 valintaa tuki myös siihen liittyvä uusi tekniikka nimeltä LINQ. LINQ oli tekniikka jota projektin alkuvaiheesta lähtien pidettiin yhtenä testauksen arvoisena vaihtoehtona.

Ratkaisuvaihtoehtojen kartoituksessa määräytyi tärkeäksi, että mukana on yksinkertainen sovellusvaihtoehto. On todennäköistä, että syntyy tilanteita, joissa on kiire saada aikaan halutunlainen resurssitehokas integraatio, jossa kokonainen selainratkaisu on luotava alusta lähtien ja aikaa tuhlaamatta. Täten toiseksi ratkaisuvaihtoehdoksi rakennettiin ASP.NET-sovellus, joka sisältää paljon valmiita .NET-komponentteja, joita voi hallita Visual Studiolla. Toisen vaihtoehdon tuli näin ollen edustaa monimutkaisempaa selainratkaisua, jonka kehitystyö on monimutkaisempaa ja aikaa vievempää. Selainratkaisuihin rakennettiin juuri luotua kriteeriä mukaillen useita vertailupareja, joiden kautta voitiin tutkia suoriutumista. LINQ-tekniikalle oli löydettävä vastakohde, johon sitä voitiin verrata. Käyttökokemukset toivat esiin Stored Procedures -tekniikan. Sitä luonnehditaan hyväksi ja resurssitehokkaaksi vaihtoehdoiksi toteuttaa tietokantahakuja SQL Server -tietokantoihin. Tekniikka siis edusti kaikkea sitä, mitä tässä projektissa olevalta tekniikalta vaadittiin. Yksi tärkeä vertailupari oli luotu. Toinen vertailupari oli graafisen- ja taulukkomuotoisen datan vertailu, joka aiheuttaa paljon keskustelua sekä ammattilaisten, että peruskäyttäjien keskuudessa. Mukaan tutkintakohteisiin otettiin myös tilanhallinta (sessio), navigointi ja välimuistinhallinta.

5.2 Testausprosessin tietokanta

Koska integrointiympäristö ei ollut vielä ajan tasalla, piti luoda vaihtoehtoinen tietokanta, jolla toimivuudet saatiin testatuksi. Tässä vaiheessa luotiin yhden taulun SQL Server Express 2005 -tietokanta. Tietokannan nimi on asiakas, jossa yksi taulu: asiakas. Kyseessä on siis suhteellisen yksinkertainen yhden taulun tietokanta. Myöhemmin vastaavanlainen tietokanta luotiin myös SQL Server 2008 -tietokantaan, joka toimi osana Microsoft Dynamics NAV -järjestelmää. Tietokannan yksilöivänä arvona eli perus-

avaimena on id, joka on lukumuotoista dataa (int). Järjestyksessään seuraava vapaa lukuarvo annetaan kentälle automaattisesti ja näin kirjatut asiakastiedot saadaan yksilöityä. Muuten kaikki sisään syötettävä data on tekstimuotoista (varchar 100).



Kuva 1. Tietokanta esittely (asiakas.mdf).

5.2.1 Selainratkaisuiden yhteiset ominaisuudet

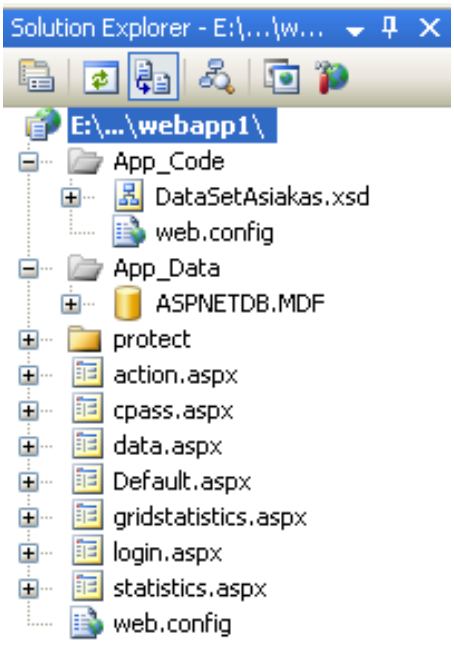
Sovelluksiin on lisätty valmiit toiminnallisuudet, joilla hallitaan salasanan vaihtoa sekä pääkäyttäjän kansiossa käyttäjätilien luontia sekä profiilin tyypin (per tai pk) hallintaa. Ne eivät kuitenkaan sisälly testattaviin kohteisiin, mutta loivat ratkaisulle ominaisuuksia, joilla voidaan hallita käyttäjäprofiileita. Lisäksi sovellusten välimuistia hallitaan yhden tietolähteen sovelluksille suunnatulla output caching -välimuistinhallinnalla. Aika-arvo on määritetty 60 sekuntiin. Tietolähde siis päivitetään minuutin välein. Tällä välillä tieto pysyy muuttumattomana internet-sivulla. Output caching soveltuu sivuille, joilla luetaan dataa tai lisätään sitä. Raja-arvoja käsiteltiin tarkemmin aiemmin tässä työssä. On kuitenkin suositeltavaa, ettei kontrolloi välimuistinhallinnalla esimerkiksi kirjautumissivua. Tällöin esimerkin mukainen kirjautuminen tapahtuisi järjestelmään vasta minuutin kuluttua aktivoinnista. Kaikki data ei ole sopivaa välimuistinhallinnan kannalta tarkasteltaessa.

5.2.2 Selainratkaisu 1: kuvaus

Ensimmäiseksi luotu ratkaisumalli kaavailtiin kuvaamaan nopeaa perusratkaisua. Selainratkaisu 1, työnimeltään webapp1, koostuu seuraavista sivuista: login.aspx, Default.aspx, action.aspx, data.aspx, statistics.aspx, gridstatistics.aspx ja cpass.aspx. Lisäksi ratkaisussa on protect-niminen kansio, jossa on sivut: controlx.aspx, Default.aspx ja control.aspx. Sovellus on määritetty käynnistyväksi login-sivulta ja kirjoittamalla oikean tunnuksen pääsee järjestelmään sisään. Default-sivulla on tekstilinkein suoritettu navigaatio, josta päästään hallinnoimaan sovelluksen peruskäyttäjän toimia. Lisäksi oikeuksien riittäessä pääsee hallintapuolelle protect-kansioon. Muilta sivuilta paluu suoritetaan tekstilinkin välityksellä, jotka tuovat takaisin Default-sivulle.

Tiedon lukemista varten selainratkaisuun on luotu sivu nimeltä data.aspx. Sivulla on graafinen esitysobjekti (ListView), jonka kautta data esitetään sivun käyttäjälle. Tietoa varten on luotu SqlDataSource. Haku suoritetaan

Stored Procedures -tekniikalla (Liite 6). Action-sivu sisältää puolestaan käyttäjän kannalta tärkeät toimet: asiakastiedon lisäys, muokkaus ja poisto. Vaikka LINQ on taustaprosessoinnissa vahvasti mukana tässä sovelluksessa (.NET 3.5 uutuus), olen toteuttanut datan käsittelyn ADO.NET-tekniikalla. Tiedonkäsittelyssä on luotava ObjectDataSource, jotta data on käsiteltävässä muodossa. ObjectDataSource hyödyntää tiedostoa DataSetAsiakas.xsd. Datasetin sisältönä on tietokannan asiakas, taulu asiakas. Data esitetään .NET 3.5-komponentilla ListView. Lisäksi sekä taulukkomainen sekä graafinen asiakkaiden lukumäärän haku paikkakunnittain on toteutettu Stored Procedures -tekniikalla (Liite 5). Tieto esitetään graafisesti (DataList) ja taulukkomuodossa (GridView). Stored Procedures on tässä tapauksessa prosessoitu SqlDataSourcen avulla, sillä tietoa vain luetaan, sitä ei muokata.



Kuva 2. Sovelluksen webapp1 rakenne.

5.2.3 Selainratkaisu 2: kuvaus

Kyseiselle ratkaisulle on kaavailtu vastaava kirjautumisjärjestelmä kuin ratkaisulle webapp1. Ratkaisussa toiminnot on eritelty niin, että käyttäjäystävällisyyttä on saatu paremmaksi. Perusrakenteeltaan ratkaisu on webapp1-sovellukseen verrattuna samantyylinen. Näin testauksessa tarpeen oleva vertailtavuus on mahdollistettu. Tietokannan dataa prosessoidaan tässä sovelluksessa uudella .NET 3.5 -tekniikalla, LINQ, sekä Stored Procedures -tekniikalla. LINQ tarvitsee toimiakseen tiedoston DataClassesAsiakas.dbml. Sinne on määritetty käytettävä tietokanta ja taulu. tässä tapauksessa tietokannan asiakas, taulu asiakas.

Navigaatio on tässä ratkaisussa toteutettu ASP.NET-toiminnallisuudella SiteMap (Liite 10). SiteMap on yksi testauksessa tutkittavista kohteista. Sitä verrataan tekstilinkkeihin resurssikuormituksen näkökulmasta. Visual Studiosta löytyy valmiita komponentteja SiteMap-navigoinnin toteuttami-

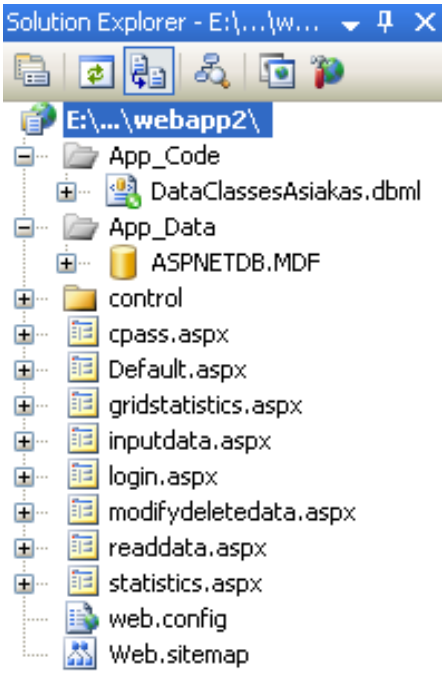
seen. SiteMap-navigaatiossa käytettävä tieto on varastoitu web.sitemap-tiedostoon.

Asiakastiedon lukeminen toteutetaan sivulta readdata.aspx. Haku prosessoidaan käyttäen VB.NET ohjelmointikieltä ja LINQ-tekniikkaa (Liite 7). Data esitetään internet-sivulla GridView'lla. Luontiprosessissa LINQ'n kanssa saatiin parhaat tulokset aikaan taulukkomuotoisella esitystavalla. Tarkoituksenmukaiset graafiset objektit hylkivät itse ohjelmoitua LINQ-käsittelyä aiheuttaen virheilmoituksia. Tämän oli oltava järjestelmävirhe.

Koska perussovellus hyödynsi LINQ-tyyppistä tiedon peruslisäystä, otettiin tässä ratkaisussa mukaan Stored Procedures. Näin saadaan vertailukohde aiemmin luodulle lisäystavalle. Aiemmassa versiossa käyttäjäystävällisyys ei ollut hyvä ja haittasi nopeaa työskentelyä. Tässä ratkaisussa on saavutettu parempi käyttäjäystävällisyys, koska jokainen komponentti voidaan asetella vapaasti. Tässä vaihtoehdossa piti lisätä koodia sekä ASP.NET-sivulle (Liite 9), että käyttää Stored Procedures -komentoa (Liite 4). Asiakastiedon muokkaus ja poisto on puolestaan toteutettu valmiilla LINQ-ratkaisuilla. Esitysobjektiksi on valittu ListView. Esitysmuoto on graafinen, mutta valmiissa toiminnallisuudessa ongelmia ei ilmennyt, kuten itse toteutetussa. Tässä vaihtoehdossa SqlDataSource on määritetty koodissa manuaalisesti.

Graafinen ja taulukkomuotoinen datan esitys asiakkaiden lukumäärän mukaan paikkakunnittain on toteutettu tässä vaihtoehdossa LINQ-tekniikalla, ohjelmoiden VB.NET-ohjelmointikielellä. Kuten aiemmin kävi ilmi, LINQ prosessoi luotua koodia huonosti, ja graafisessa esityksessä oli löydettävä vaihtoehto, joka toimii ja on kevyt. Sopivaksi vaihtoehdoksi osoittautui BulletedList. Taulukkomuotoinen datanesitys suoritettiin vastaavalla tavalla käyttäen tiedon esitykseen GridView-taulukkoa (Liite 8).

Sovellukseen on myös lisätty evästeellinen sessio. Tätä varten ohjelmakoodiin on lisätty Default-sivulle koodia (Liite 11) ja lisäksi web.config tiedostoon <sessionState cookieName="NAV" mode="InProc" />. Koodissa luodaan sessio, joka käynnistetään web.config-tiedostossa. Tämä ei ole oletusarvoisesti päällä. Nimeksi evästeelle on annettu ”NAV”. Evästeen toimivuus tarkistettiin ja todettiin toimivaksi.



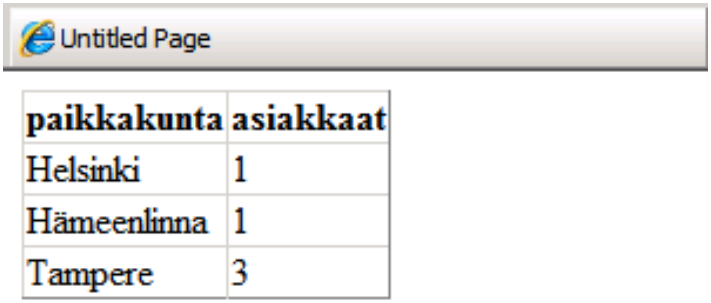
Kuva 3. Sovelluksen webapp2 rakenne.

5.2.4 Selainratkaisut ja integrointi Microsoft Dynamics NAV'iin

Tässä projektissa molemmat selainratkaisut integroitiin Microsoft Dynamics NAV 2009 -järjestelmään käyttäen välityskappaleena SQL Server 2008 -tietokantaa. Tämä on myös resurssitehokkuuden näkökulmasta erinomainen tapa, sillä näin suora liitos on nopeampi kuin integrointi suoraan järjestelmään. Suorat NAV-integraatiot ovat resurssikulutukseltaan suurempia. Asiaa testattiin tämän projektin puitteissa käyttämällä vertailumittarina NAV'ssa jo integroituna ollutta NAV-clientia.

Integroinnin alussa varmistettiin, että järjestelmässä on asianmukainen tietolähde. Ensin kirjaudutaan NAV-palvelimelle ja luotiin sinne asiakasniminen SQL Server 2008 -tietokanta, jossa on taulu asiakas. Alkuun ongelmia tuottivat häiriötekijät, joissa tietokantaan otti taustaprosessi yhteyttä. Tietokannan kanssa asioivat prosessit oli suljettava, ennen kuin tietokanta voitiin liittää sovellukseen. Seuraavaksi suoritettiin liitos Visual Studiossa.

Seuraavaksi demontroidaan NAV-integraatio käytännön esimerkillä. Ensin avataan Visual Studio 2008 Professional (Visual Web Developer -osa) ja avataan esiin selainratkaisu, johon integraatio toteutetaan. Etsitään Tools-välilehti ja sieltä Connect to Database. Yhdistetään kyseinen tietokanta sovellukseen ja hyväksytään valinta. Tieto yhteysmerkkijonosta tallentuu tiedostoon web.config, kohtaan <connectionStrings>. Kun selainsovellus kykenee käsittelemään tietokantaa, integraatio toimii. Alla on kuva tietokantahausta integraation jälkeen.



Kuva 4. Stored Procedures ”paikkakunnittain” internet-sivulla (IE) onnistuneen integraation jälkeen.

Kyseinen tapa integroida osoittautui resurssitehokkaaksi ja oli tässä projektissa hyvä vaihtoehto. Suora integraatio tietokantapalvelimeen on nopea toteuttaa ja nopea prosessoimaan tietoa. Yleisesti integraatioita voidaan tehdä monella tapaa, mutta tässä projektissa virtuaalialustat tuottivat paljon ongelmia. Uusien palvelinkoneiden yhdistäminen järjestelmään olisi ollut riski. Lisäksi tässä ratkaisussa käytettävä tietolähde on pieni. Ei ole resurssitehokasta kierrättää dataa useiden palvelimien kautta, ellei siihen ole syytä.

5.3 Selainratkaisujen testausprosessi

Toimeksiantoon kuului asianmukaisen testauksen suunnitteleminen ja toteutus. Tarkoituksena oli määrittää selainratkaisujen resurssitehokkuus ja suoriutumisen rajat. Lisäksi tarkoituksena oli saada tietoa sovelluksissa käytetyistä komponenteista ja niiden resurssikuormituksesta. Selainratkaisujen vertailu oli tärkeä kriteeri, joten testauksen tulosten oli oltava muodossa, jossa voitiin osoittaa keskinäinen suoriutumisero toteen. Tilanteessa, jossa selainratkaisut asetetaan toisiaan vastaan, saadaan aikaan dataa, jossa voidaan osoittaa suoriutuvuuserot. Saadulla datalla voidaan lisäksi osoittaa se, mitä testattuja komponentteja käyttämällä päästään suoritustehokkaimpaan resurssitasoon. Tietoa voidaan hyödyntää tulevaisuudessa sovelluskehityksessä. Testauksen tarkoitus on nykyhetken lisäksi myös kartoittaa tulevaa. Sovelluskehitys on jatkuva prosessi. Kehitettävää löytyy aina.

5.3.1 Testausympäristö

Testauksessa käytetty testausalusta koostuu Hämeen ammattikorkeakoulun virtuaalipalvelimella olevista virtuaalikoneista, jotka sijaitsevat konfiguraatiossa nimeltä config1918. Testausalusta oli lainassa toiselta projektilta, joten sillä oli valmiiksi kuormaa. Se ei ollut puhdas. Vaikka tämä haittasi testaamista, ei se estänyt saamasta aikaan päätuloksia. Tulokset osoittavat silti vertailtavuuseron, mutta data ei ole tarpeeksi eksaktia, jotta siitä voisi määrittää yksityiskohtaisia linjoja. Tulokset ovat ongelmista huolimatta vertailukelpoisia. Koska kyseinen testausympäristö ei ole todennäköinen alusta kyseisille sovelluksille tositilanteessa, ei alustan absoluuttisilla arvoilla ollut suurta merkitystä. Keskinäinen vertailu voitiin kuitenkin toteuttaa suunnitellusti.

Testausalusta koostui kaiken kaikkiaan viidestä virtuaalikoneesta: config1918VM0, config1918VM1, config1918VM2, config1918VM3 ja config1918VM5. Niistä virtuaalikone config1918VM0 on DC (domain controller), eli tältä koneelta hallitaan toimialuetta. Muut testauksessa käytössä olevat virtuaalikoneet ovat: config1918VM1 (NAV-client), config1918VM2 (SQL Server -palvelin, jolla sijaitsee toimialueen tietokannat) sekä config1918VM5 (NAV 2009). Yksi virtuaalikone, config1918VM3, on tarpeeton ja se pidetään testauksen ajan suljettuna.

Kyseisen projektin näkökulmasta palvelinalusta toimi seuraavaan tapaan. config1918VM0 pyöritti toimialuetta, eikä liittynyt testiin muulla tapaa. Testin luonteen ja aikataulujen vuoksi palvelinympäristön kykyä ei testattu. Virtuaalipalvelimelle, config1918VM1, on siirretty selainratkaisut. Tietokannat sijaitsevat palvelimella config1918VM2, sinne asennettiin Visual Studio 2008 Professional Edition. Sitä käytettiin tietokantayhteyksien luomiseen. Lisäksi kyseinen sovelluskehitin toimitti web-palvelimen roolia. Visual Studioissa on sisäänrakennettu IIS. NAV-palvelin, config1918VM5, kommunikoi tietokantapalvelimen kanssa.

5.3.2 Testaussuunnitelma

Jotta testaus voitiin toteuttaa asianmukaisesti, tarvittiin suunnitelma. Tässä projektissa päätettiin toteuttaa kuormitus- ja rasitustestaus. Testaus perustuu aiemmin opinnäytetyössä esiteltyihin konsepteihin. Konsepteja sovellettiin, jotta testit vastaavat kyseisen projektin tarpeita, ja ovat rakenteeltaan sellaisia, että niistä saadaan muodostettua tarvittava data tarpeellisen vertailuanalyysin tekemiseksi. Ylimääräisen datan tuottamista pyrittiin välttämään. Testauksen avuksi luotiin työkuormamallilomake (Liite 3), Johon tärkeimmät kriteerit kirjattiin.

Ennen testausta luotujen suunnitelmadokumenttien lisäksi oli tarpeen määrittää testauksessa käytettävät mittaus ja valvontatyökalut sekä lisärasitusta tuottava apusovellus. Testauksen luonteen vuoksi mittausvälineiden tärkein tarkoitus oli olla molemmille osapuolille tasapuolinen, eikä tarpeen ollut luoda täsmällistä dataa. Riitti, että sovellusten välinen vertailu voitiin suorittaa niin, että jokaista mitattavaa kohdetta voitiin arvioida asianmukaisella arvolla vastakohteeseensa.

Vertailu päätettiin toteuttaa tässä projektissa RAM-arvokuormituksiin sekä prosessoritehon kuormituksiin pohjautuen. RAM-muistin suorituskykyä suhteutettiin prosenttiyksikkömuutoksien avulla määrittämällä prosenttiarvon muutos alkutilanteesta kohonneeseen arvoon. Mittarin oli oltava sellainen, että se tiedostaa RAM-muistin määrän, osaa laskea jo käytössä olevan RAM-muistin määrän ja kykenee osoittamaan kuormituksen määrän kasvun loppuolosuhteessa. Työkaluksi päätettiin valita Windows Task Manager. Prosessoriteho ja sen muutokset analysoitiin myös Task Managerin avulla. Mittarin tässä testauksessa tuli osoittaa prosessorikulutuksen määrä. Lisäksi sen piti osata määrittää jo käytössä olevan prosessorikulutus ja kyetä osoittamaan kulutuksen kasvu loppuolosuhteessa. Kuormitustason kasvattamisessa käytettiin apuna avoimen lähdekoodin kuormitusso-

vellusta, HeavyLoad 3.0. Kuormitussovellusta hyödynnettiin kuormitustestauksessa resurssikulutuksen kasvun simulointiin sekä luomaan kasvanutta rasitusta palvelimille rasitustestauksen aikana. Lisäksi testauksen yhteydessä arvioitiin ratkaisuiden suoriutumisaikaa. Testauksen aikarajaksi asetettiin kolme sekuntia, jonka aikana sovelluksen oli suoriuduttava kaikista avainskenaarioista. Kolmen sekunnin aikaraja toimi määreenä, jonka alitusta pidettiin tärkeänä kriteerinä ja ylitystä pidettiin kriittisenä pisteenä. Kolmensekunnin aikarajan ylityksen jälkeen tehokkuus oli kärsinyt liikaa ollakseen suoritustehokas. Tällöin testaus voitiin kyseiseltä osalta päättää.

Jotta testauksesta saatiin paras mahdollinen hyöty irti, oli määritettävä testauksen tavoitteet ja testauksen mittausmääreet. Testattavat kohteet oli valittava niin, että tuloksia vertailemalla saadaan todennettua sovellusten suoriutumiskyky asetetuista tavoitteista sekä määrittää selainratkaisuiden keskinäinen paremmuus. Lisäksi sovelluksia piti kuormittaa, jotta saatiin esille niiden mahdolliset heikot kohdat, sekä hankkia dataa siitä, kuinka sovellukset toimivat poikkeuksellisissa olosuhteissa runsaan rasituksen alla tai kasvaneissa resurssitarpeissa.

Kyseistä projektia varten toteutettu kuormitustestaus suunniteltiin kaksivaiheiseksi. Ensin pyrittiin mittaamaan sovelluksen yleinen kuormitustaso, jonka kautta selvitettiin RAM-muistin ja prosessoritehon käyttö kussakin määritetyssä avainskenaariossa. Näin saatiin luotua dataa, jonka avulla voitiin analysoida kyseenomaisen selainratkaisun suoriutumista asetetusta tavoitteesta.

Rasitustestauksessa pyrittiin nostamaan kuormitustasoja niin, että saatiin selville sovellusten käyttäytyminen kasvaneen rasituksen alla. Rasitustestauksesta saadun datan turvin pyrittiin selvittämään pisteitä, joissa sovellus on selkeästi hidastunut. Lisäksi rasitustestauksen kautta voitiin simuloida tilanteita, joissa kuormituksen määrä vastasi palvelunestohyökkäystä tai suurta päivitystilannetta, jossa palvelinympäristön suorituskyky on kohonnut normaalista. Lisäksi rasitustestauksen kautta pyrittiin varmistamaan sovelluksien suoritusvarmuus. Verkkosovellukselle on tärkeää, ettei se petä tositilanteessa. Rasitustestaus suoritettiin sovellusrasitustestauksena. Pienessä sovelluksessa virheet oli mahdollista määrittää tarkasti ilman tarkempaa testausta.

5.4 Testausraportti: eteneminen ja tulokset

Selainratkaisuille webapp1 ja webapp2 suoritettiin resurssikuormitus testaus. Testaus pohjautui aiemmin esitettyihin konsepteihin, jotka määritettiin ennen testausprosessin käynnistämistä. Testaus perustui arvojen keskinäiseen vertailuun, ei absoluuttisen datan tuottamiseen. Testauksen luonteen vuoksi esitetyt arvot olivat epätarkkoja sekä pohjautuvat laina-alustan toimintaan eli arvoista voidaan näyttää toteen keskinäinen suoriutuvuus, muuten hankittu data on epärelevanttia. Mikäli olisi haluttu saada täsmällisempää, absoluuttisesti paikkansapitävää, dataa, pitäisi myös mittauskalusto hienosäätää ja kalibroida ennen prosessin alkua. Seuraavaksi esitetään suoritettujen kuormitus- ja rasitustestausten kautta saatu olennainen data.

5.4.1 Kuormitustestaus

Kuormitustestauksen ensimmäisessä vaiheessa webapp1 ja webapp2 asetettiin normaalitilannetta vastaavaan kuormitustasoon. Testaus alkoi lähtötasojen kartoituksella. Testauksessa käytin SQL Server -palvelinta myös selainratkaisuiden käyttöpalvelimena. Valintaan vaikutti laina-alustan rakenne sekä se, että kuormitus määritettiin tällä tavoin korkeammalle. Tämä oli testauksen kannalta kriittisin palvelin, sillä siellä sijaitsi käytettävä tietokanta. Toinen kriittinen palvelin oli NAV-palvelin. Alkuperäisestä suunnitelmasta poiketen päätettiin myös jakaa selainratkaisut verkkoon Visual Studion sisäisellä IIS-palvelimella. Täten aikaa ei kulunut webpalvelimen pystyttämiseen.

Perustasossa palvelimen RAM-muistin peruskulutustaso oli jo lähtökohtaisesti arvossa 44 %. Visual Studio nosti kuorman arvoon 56 %. Internet Explorer -selaimen käynnistys nosti kulutuksen arvoon 61 %. Sisään kirjautunut käyttäjä nosti selainratkaisussa webapp1 kulutuksen arvoon 62 % sekä perus- että pääkäyttäjäprofiililla. Käyttäjien määrän lisääminen nosti RAM-kulutusta prosenttiyksiköllä kirjautunutta käyttäjää kohden. Selainratkaisu webapp2 nosti kuormituksen kirjautumisen yhteydessä arvoon 63 %. Käyttäjätilien lisääminen nosti kuormitusta prosenttiyksiköllä kirjautunutta käyttäjää kohden. Tästä voidaan päätellä se, että selainratkaisussa webapp2 oleva evästeellinen sessio varaa RAM-muistia prosenttiyksikön verran tässä järjestelmässä. Prosessorikulutuksessa sisään kirjautuminen kohotti molemmilla ratkaisuilla kulutusta pikaisesti arvoon 70 % ja uloskirjautuminen varasi prosessoritehosta 80 % prosessoidessa poistumista. Sisään kirjautuminen ja kirjautuminen ulos olivat selkeästi kuormittavimmat prosessit, jolloin molemmat ratkaisut toimivat hitaimmin.

Seuraavaksi kuormitustestauksessa edettiin kohti avainskenaariotestausta, jossa pyrittiin mittaamaan kulutusarvot kriittisimmiksi määritetyissä tilanteissa. Webapp1 suoriutui niistä seuraavasti. Tiedon syöttö ja muokkaus nostivat prosessoritehoa korkeimmillaan arvoon 14 %. Tiedon poistoa suorittaessa noustiin arvoon 15 %. Graafinen Stored Procedures -tiedonhaku kohotti kuormitustason arvoon 10 % ensimmäisellä kerralla. Toisella hakukerralla kuormitus laski arvoon 7 %. Taulukkomuotoinen datanesitys kulutti prosessoria arvolla 11 %, toisella kerralla 7 %. Tiedon haku samaisella tekniikalla toteutettuna kulutti resursseja 11 % ja toisella kerralla 7 %. Kuormitustestauksen ensimmäisessä vaiheessa toinen ratkaisu, webapp2, suoriutui vastaavista toimista seuraavaan tapaan. Tiedonsyöttö Stored Procedures -tekniikalla koodaten niin, että tietokantayhteys otetaan prosessointihetkellä, kuormitti prosessoria vain arvoon 7 %. Tiedon LINQ-poisto ja muokkaus pyörivät 15 % tuntumassa. LINQ-tekniikalla toteutetut haut: perushaku, graafinen erikoishaku sekä taulukkomuotoinen haku tuottivat seuraavat arvot. Perushaku ensimmäisellä kerralla 16 %, toisella 7 %. Graafinen esitys ensimmäisellä kerralla 15 %, toisella 8 %. Taulukkoesitys otti ensin 21 %, toisella kerralla 8 %.

Havaitsemisen arvoista on se, että toisella toteutuskerralla arvot ovat alhaisemmat. Tähän on syynä tietokoneen välimuisti. Kun välimuistia hallitaan, saadaan aikaan resurssisäästöä, sillä tieto sivulla ei päivity automaattisesti kokoajan. Välimuistin hallinta aiheuttaa kuitenkin kuormituspiikke-

jä päivityshetkinään. Tästä syystä välimuistiin ei saa varastoida liikaa dataa. Kun jonossa oleva data päivitetään järjestelmään, syntyy suuri kuormituspiikki, joka pahimmillaan vahingoittaa järjestelmää aiheuttaen kaatumispisteen ylittymisen. Kyseinen testaus sisältyy kapasiteettitestaukseen, jota ei suoritettu tämän projektin puitteissa.

Seuraavaksi siirryttiin eteenpäin uuteen vaiheeseen, joka oli käyttöaikaarvojen määrittäminen. Molemmat ratkaisuvaihtoehdot suoriutuivat kaikista avainskenaariotoimista noin sekunnissa. Webapp1 oli aavistuksen nopeampi toimissaan. Webapp2 navigointijärjestelmineen oli aavistuksen hitaampi. Tälle testaukselle suoritettiin myös toinen vaihe, jossa kuormitusta lisättiin HeavyLoad 3.0 -sovelluksen avulla, jotta saatiin simuloitua tilanne, jossa käyttäjämäärä on kasvanut. RAM-kuorma nostettiin noin 80 %:iin, prosessorikuormitus noin 30 %:iin ja kiintolevyn kirjoitusjono nostettiin 0,01 sekunnista 0,05 sekuntiin. Sovellukset toimivat edelleen puolentoista sekunnin raja-alueella. Ottaen huomioon palvelinalustan kuormitustason ja simuloidut korotukset, voidaan tulkita että käyttäjämäärät voivat kasvaa ja selainratkaisut ovat edelleen suoritustehokkaita näissäkin olosuhteissa. Kuormitustestauksen seuraavassa vaiheessa suoritettiin tiedonsyöttötesti, jossa sisään kirjautuneella profiililla syötettiin tietoja järjestelmään. Molemmat ratkaisut suoriutuivat testistä hyvin, eikä ongelmatilanteita syntynyt. Suoriutumisaikoihin ei ilmennyt muutoksia tässä yhden ihmisen suorittamassa kuormitustestauksessa.

Kuormitustestauksen tuloksista voi päätellä seuraavaa. Analysoidessa saatua dataa on huomioitava se, että prosenttiarvot ovat mitatut laitteilla, joiden luotettavuus ei ole testattu. Saatu data perustuu täten epätarkkoihin arvoihin. Tästä huolimatta testauksen kautta saatiin aikaan tuloksia, jotka ovat käytännöllisiä selainratkaisujen keskinäisen vertailun näkökulmasta. Päähuomioina mainittakoon, että tulokset todistavat välimuistinhallinnan merkityksen resurssikuormitukseen. Lisäksi evästeellinen sessio ei ole kevyelle selainratkaisulle resurssitehokkuutta kehittävä asia. Evästeellä on oma kokonsa ja se syö resursseja kokonsa verran. Prosentuaalisesti eväste syö kevyeltä selainratkaisulta enemmän kuin suurelta. Lisäksi Stored Procedures osoittautui LINQ-tekniikkaa resurssitehokkaammaksi. On suositeltavaa käyttää sitä silloin kuin mahdollista. Graafinen tiedonesitys on resurssitehokkaampaa kuin taulukoiden käyttäminen. Välimuistinhallinnalla tätä väliä saadaan kurottua. Lisäksi testit havainnoivat, että tyhjä graafinen esitys on paljon keveämpi kuin tyhjä taulukko. Kun tietoa syötetään lisää, alkavat arvot tasoittua. Kun järjestelmään oli lisätty kolmekymmentä asiakastietoa, kuormitustasojen väli oli kuroutunut yhteen prosenttiyksikköön.

5.4.2 Rasitustestaus

Kuormitustestauksen valmistuttua siirryttiin rasitustestaukseen. Rasitustestauksessa pyrittiin selvittämään selainratkaisujen hidastumisvauhtia. Lisäksi testauksen tarkoituksena oli osoittaa käyttövarmuus kohonneissa rasitusarvoissa. Kohotetut rasitusarvot simuloivat tilannetta, jossa on tapahtunut jotain odottamatonta: järjestelmä on palvelunestohyökkäyksen kohteena, rasitustasot ovat kasvaneet laitevian, käyttäjäpiikin tai suuren ohjelmistopäivityksen takia. Lisäksi testillä halutaan varmistaa, ettei ratkai-

sun suorituskyky petä, kun rasitusarvot ovat korkealla. Testauksen kannalta testissä pidettiin kriittisenä sitä, että selainratkaisu pysyy kolmen sekunnin raja-arvon sisäpuolella. Lisäksi RAM-kuormituksen ylärajaksi määritettiin 90 % ja prosessoritehosta käytössä sai olla korkeintaan 80–85 %. Kokeellisuuden ja testaussovelluksen kontrolloinnin vuoksi tarkkaa arvoa ei voitu määrittää.

Webapp1 oli niin vakuuttava kuormitustestauksessa, että ajankäytöllisistä syistä päädyttiin testaamaan sovellus lähtien liikkeelle yläraja-arvosta. Kuormitus nousi yläkriteerille ja siitä huolimatta sovellus prosessoi viiden minuutin session ilman ongelmia. Toimet saatiin toteutettua noin kahdessa sekunnissa skenaariota kohden. Jotta raja-arvoihin päästäisiin ja saataisiin testattua toimivuusvarmuutta, siirryttiin kuormittamaan NAV-palvelinta. Tällä ei ollut vaikutusta prosessointitehoon integraation luonteesta johtuen. Lisäksi tässä kohtaa suoritettiin varjotesti, jossa kokeiltiin NAV:n kanssa toimivaa NAV-client-sovellusta. Tässä tilanteessa NAV-client hyytyi täysin. Tässä vaiheessa myös virtuaalipalvelimen client menetti yhteyden NAV-palvelimeen ja ruutu pimeni. Itse palvelin ei silti lakannut toimimasta. Testausta jatkettiin kuormittamalla kokeellisesti tietokantapalvelinta samalla tavalla. Nyt ylitettiin kolmen sekunnin arvo ja testaus päättyi siihen. Huomion arvoista on, että sovellus kesti käytössä hyvin, eikä mitään jäänyt suorittamatta, oli rasitusta kuinka paljon tahansa. Käyttäjän kannalta webapp1 oli suoritusvarma ja nopea. Suoritin saman testin myös webapp2-sovellukselle. Sen suoritusvarmuus oli samaa luokkaa. Webapp2-ratkaisu ylitti aavistuksen aikaisemmin kolmen sekunnin rajan, mutta vasta samassa testausvaiheessa. Määritettyjen testausarvojen puitteissa myös kyseinen sovellus oli hyvä.

Integraatiotapa osoittautui erittäin suoritustehokkaaksi. Etsittäessä suoritustehokasta integrointivaihtoehtoa kevyille selainratkaisuille, on liitos SQL Serverin kautta suositeltava tapa. On kuitenkin huomionarvoista, että tässä projektissa kyseiseen päätelmään on tultu resurssitehokkuuden näkökulmasta. Asianmukaiset tietoturva-analyysit on myös syytä toteuttaa ennen tuotantokäyttöön siirtymistä.

6. YHTEENVETO

Tämä opinnäytetyöprosessi on ollut haastava ja palkitseva: työn kautta olen oppinut paljon uutta, mutta prosessin edetessä koin lukuisia takaiskuja. Negatiivisetkin asiat on mahdollista kääntää voimavaraksi, kunhan pitää mielen virkeänä ja jaksaa uskoa itseensä. Projektiin lähtiessä tietotasoni Microsoft Dynamics -tuotteista olivat vähäiset. Laajemmalla kokemuksella olisin tiennyt paremmin mihin olen ryhtymässä ja osannut sopimusta tehdessä rajata työtä paremmin. Jos aloittaisin saman projektin nyt uudestaan, tekisin monia asioita toisin. Osaisin paremmin hahmottaa mitä työhön voidaan sisällyttää ja mitä ei. Lisäksi osaisin määrittää paremmin miten toimitaan kun tekniset ongelmat haittaavat prosessia liikaa. Osaisin vaatia määrätietoisemmin myös käyttööni kalustoa, joka toimii. Nyt paljon aikaa tuhraantui siihen, että saamani testausalustat ja integraation kohde (NAV) eivät olleet toimintakuntoisia. Virtuaalipalvelimet haittasivat työni etenemistä alusta loppuun. Olen onnellinen, että projekti on valmis ja täyttää toimeksiannon määrittämät vaatimuksensa. Projekti on ollut myös laaja ja prosessi on kehittänyt minua niin resurssitehokkaiden verkkopalveluiden kehittäjänä, sovellustestaajana kuin NAV-asiantuntijana. Opinnäytetyöprosessi on täten kehittänyt montaa osaamisaluetta. Etenkin testaaminen ja NAV olivat täysin uusia asioita.

6.1 Henkilökohtaisten tavoitteiden asettaminen

Kun aloin pohtia itselleni opinnäytetyöaihetta, päädyin nopeasti linjaukseen, jossa mukana on .NET Framework ja verkkopalvelut. Sitä kautta aloin jäsentää mielessäni resurssitehokkuutta mukaan aiheeseen opinnäytetyön ohjaajani avustuksella: resurssitehokkuus on mutkikas asia ja yksi tärkeimmistä tekijöistä verkkopalveluissa ja niiden toimivuudessa. Aiemman osaamisen kautta ASP.NET oli ennalta tuttu ja tilauksen luonteen vuoksi ASP.NET oli erinomainen vaihtoehto sovelluksen tyypiksi. Kun aiheeni jalostui Hämeen ammattikorkeakoulun toimeksiannon mukaiseksi NAV-integraatioksi, asetin itselleni tavoitteeksi oppia myös mahdollisimman paljon NAV-tuotteesta ja integraatioista. Integraatiot ovat tärkeä tekijä nykyaikaista sovelluskehitysprosessia. Kun lukee alaa käsitteleviä uutisia, integraatiot ovat aktiivisesti esillä. Opinnäytetyössäni integraatioiden osuus on myös tärkeä, päähuomio on resurssitehokkuudessa.

Aiheeseen liittyvä materiaali on ollut helppoa hankkia. Kirjallisia lähteitä tarvittiin vähän ja tekninen materiaali oli suureksi osaksi Microsoftin tarjoamana verkossa. Lisälähteitä hain aihetta käsittelevistä alan asiantuntijoiden materiaaleista. Moniin muihin töihin verrattuna materiaalin hankinta on ollut suhteellisen helppoa. Projektin alkuvaiheessa kartoitin muitakin lähteitä, mutta niistä ei ollut saatavissa lisäarvoa. Varsinkin kirjalliset lähteet olivat erittäin kattavat ja teosten kokonaissivumäärä on yli 2000 sivua. Valinnat ovat perusteltuja työn luonteen vuoksi, Microsoft ja .NET, tuloksesta ei ole tarvinnut tinkiä lähteiden vähäisyyden vuoksi, sillä lähteet ovat olleet erinomaiset.

6.2 Opinnäytetyön ajankohtaisuus

Kyseinen aihe on monelta mittapuulta katsottuna erittäin ajankohtainen. Toki osin työn voi nähdä toisellakin tapaa. Ajankohtaisuutta puoltaa se, että se käsittelee verkkopalveluita, jotka ovat nykyään suuri tekijä länsimaisen ihmisen arkea. Resurssitehokkuus on yksi verkkopalveluiden avaintekijöistä ja viimeaikoina on voinut mediaa seuratessaan todeta, ettemme elä täydellisessä maailmassa resurssikuormituksenkaan suhteen. Integraatioilla on myös sovelluskehityksessä suuri rooli, eivätkä ammattilaisetkaan aina onnistu siinä. Kun laitteistointegraatio tai resurssiteho pettää, ovat seuraukset ikäviä ja pitkäaikaisia. Toiminnanohjausjärjestelmillä on myös suuri rooli tänä päivänä. Niiden osaaminen on eduksi työmarkkinoilla. Microsoft Dynamics NAV on yksi runsaasti käytetty talouden- ja toiminnanohjausjärjestelmä. Toki muitakin vaihtoehtoja on tarjolla joka lähtöön, mutta myös tältä osin ajankohtaisuus täyttyy.

Ainoa kyseenalainen, joskaan ei absoluuttisen kriittinen kohde, on .NET Framework -versio. Prosessi on ollut vireillä kauan ja aloittaessani versio 3.5 oli uusi. Nyt versio 4.0 on tullut käyttöön ja sovelluksen alkavat hyödyntää sitä enenevässä määrin. Kun 3.5 oli uusin versio, 2.0 oli edelleen runsaassa käytössä. 2.0 ei ole vieläkään täysin aikansa elänyt, sillä moni käytössä edelleenkin oleva tekniikka on ollut versiossa kaksi mukana, esimerkiksi hyvin testauksessa pärjännyt Stored Procedures. Lisäksi käyttäjät eivät päivitä automaattisesti uuteen version sellaisen tullessa markkinoille. Joskus liian aikainen päivittäminen voi kostautua, sillä silloin tällöin uusi versio todetaan käyttöönoton yhteydessä vääräksi. Muistellaanpa vaikka Microsoftin Windows-käyttöjärjestelmäuudistusta muutaman vuoden takaa. Windows XP sai lisäaikaa, koska uusi tuote, Windows Vista, ei ollutkaan parempi.

6.3 Projektin tulokset, toimeksiantajan saama hyöty ja henkilökohtainen hyöty

Kaiken kaikkiaan raportti saaduista testaustuloksista osoittaa, että tulokset ovat onnistuneita. Saatu data ei mullista alaa, mutta saadut tulokset paljastavat hyödyllistä tietoa resurssitehokkaisiin ASP.NET-sovelluksiin liittyen. Lisäksi projektin aikana löytyi resurssitehokas tapa integroida ASP.NET 3.5 -sovellus Microsoft Dynamics NAV’n kanssa. Projektia voi pitää onnistuneena.

Olen varma että, opinnäytetyöni on kattava ja tarjoaa opettavan paketin resurssitehokkuudesta. Vaikkakin työ on rajattu koskemaan vain ASP.NET-sovelluksia SQL Server -tietokantojen kanssa, työn työkuorma on ollut suuri. Lisäksi työn aikana ilmenneet ongelmat ovat syöneet turhia resursseja, eikä kaikkia integraatiotapoja ole ollut mahdollista ottaa mukaan. Myöskään virtuaalipalvelimet eivät osoittautuneet ihanteelliseksi kohteeksi integraatio- ja testausalustalle.

Opinnäytetyöprosessin aikana olen henkilökohtaisesti oppinut paljon uutta. En ole pyrkinyt päästämään itseäni helpolla ja olenkin pitänyt tärkeänä, että saan tästä projektista teoreettista ja toiminnallista oppia tulevaan. Työn mitoittaminen muun elämän ja opintojen viereen ei ole aina helppoa ja suuri projekti opettaa paljon myös kirjoittajasta itsestään. Varsinkin ongelmien kasautuessa on tehnyt mieli useammin kuin kerran antaa olla. Pohjahetkinä kaikki on tuntunut ylitsepääsemättömältä, mutta en ole luovuttaja. Halusin saada aikaan tuloksia ja sain niitä. Projekti on tuottanut paljon henkistä kasvua tiedon mukana.

6.4 Ongelmat

Vaikka materiaalin hankinta oli helppoa ja osin vaivaa säästyi, kokonaisuutena työhöni liittyvä tilausprojekti ei ole ollut alkuunkaan ongelmaton. Itse sovellukset on ollut helppoa luoda erillään kotikoneella. Kuitenkin testausalustaan liittyvät lukuisat ongelmat ja toimimattomuudet ovat haitanneet työn tekemistä. Samalla työn alkuperäisessä aikataulussa pysyminen on ollut ongelmallista. Testausalustan toimintaongelmista johtuen alustat eivät ole olleet koko projektin ajan saatavissa tai saatavissa sellaisena, että sen tuotteilla olisi voinut tuottaa tarvittavia tehtäviä. Loppua kohden ongelmat alkoivat lievetä ja projekti saatiin kontrolliin. Eniten asiassa harmittaa se, että noin viikon kestävä testaussessio viivästyi tästä johtuen pitkiä aikoja. Olen tehnyt työtä kuitenkin kouluprojektina, en työnä. Muu elämä ja muut opinnot ovat olleet samalla viivalla läpi projektin.

LÄHTEET

Developing High-Performance ASP.NET Applications. n.d .MSDN. Viitattu 18.12.2011.
http://msdn.microsoft.com/en-us/library/5dws599a%28v=VS.80%29.aspx#StateManagement

Howard, R. 2005.10 Tips for Writing High-Performance Web Applications. MSDN Magazine January (1). Viitattu 1.3.2010.
http://msdn.microsoft.com/en-us/magazine/cc163854.aspx

Inside .NET Ohjelmointi. Richter, J. Suom. Juha Salmela. Helsinki: IT Press. 2003.

Meier, J, Vasireddy, S, Babbar, A & Mackman, A. 2004. Improving .NET Application Performance and Scalability. Chapter 16 — Improving ASP.NET Performance 2004. pdf-tiedosto. Viitattu 18.12.2011.
http://msdn.microsoft.com/en-us/library/ff647788.aspx

Performance Testing Guidance for Web Applications. Meier, J, Farre, C, Bansode, P, Barber, S & Rea, D.2007. Viitattu 18.12.20011.
http://perftestingguide.codeplex.com/releases/view/6690

Professional ASP.NET 3.5 In C# and VB. Evjen, B, Hanselman, S & Rader, D. Indianapolis: Wiley Publishing, Inc. 2008.

LIITE 1

TOIMEKSIANTO

1. YLEISTÄ

Microsoft Dynamics Nav on keskisuurille ja kasvaville yrityksille kehitetty talouden- ja toiminnanohjauksen järjestelmä. Järjestelmällä on yli miljoona käyttäjää yli 65 000 yrityksessä ja yli 150 maassa.

(läde:http://www.microsoft.com/finland/dynamics/nav/default.mspx)

Useille toiminnanohjausjärjestelmille on tyypillistä, että niiden käyttö internet-selaimen kautta ei onnistu tai on hidasta. Tämän vuoksi järjestelmiin on tehty historian aikana integraatioita monenlaisilla välineillä, useiden eri tahojen toimesta. Kyseisten integraatioiden laajuus ja laatu on ollut hyvin vaihteleva.

Microsoft Dynamics Nav -järjestelmä on ollut HAMK:ssa opetuskäytössä syksystä 2009 asti ja sen käyttöä on tarkoitus jatkossa laajentaa uusille opintojaksoille. Tämän vuoksi on tarpeen selvittää, kuinka järjestelmäintegraatio kannattaa toteuttaa siten että tiedon esittäminen www -sivulla olisi mahdollisimman tehokasta. Tätä tietoa hyödynnetään muun muassa uusien harjoitustöiden suunnittelussa ja oppimateriaalin luonnissa.

2. VAATIMUKSET JA LOPPUTULOKSET

Toimeksiannon lopputuloksena syntyy selvitys siitä, kuinka Microsoft Dynamics NAV -integraatio kannattaa toteuttaa Internet Explorer -selaimessa ajettavalle, .Net- teknologian avulla toteutetulle www- sivulle. Asian testaamiseksi käytännössä luodaan vähintään kaksi kilpailevaa ratkaisua, joiden tehokkuutta vertaillaan. Sivujen toiminnallisuuden tulee sisältää asiakkaiden lisäyksen, päivityksen ja poiston. Lisäksi www- sivuilta tulee löytyä raportointiominaisuus, jonka avulla asiakkaiden lukumäärä paikkakunnittain voidaan nähdä sekä graafisessa että taulukko -muodossa.

Selainratkaisujen tehokkuuden mittaamiseksi tulee tehdä testisuunnitelma, jonka mukaan testaus suoritetaan. Käytetyt menetelmät ja testitulokset ja esitetään selvityksen yhteydessä.

3. TOTEUTUSYMPÄRISTÖ

Toteutus tehdään Hämeen Ammattikorkeakoulun virtuaaliympäristöissä, jonne luodaan vähintään yksi palvelinympäristö sekä tarvittava määrä muita ympäristöjä.

Kehitys tapahtuu Hämeen Ammattikorkeakoulun kehitysympäristöissä, joista löytyvät .Net-sovellusten luontiin tarvittavat työvälineet.

4. LOPPUTULOSTEN ARVIOINTI

Lopputuloksia arvioidaan selvitystyön laajuuden ja monipuolisuuden perusteella. Erityisiä tarkastelun kohteita ovat www -sivujen ratkaisuvaihtoehdot, testien laajuus sekä suunnitelmallisuus.

LIITE 2

TESTAUSALUSTAN DOKUMENTAATIO

Nimi:
config 1918VM5

Domain:
mattikatti.fi

Palvelin:
Windows Server 2008 R2 Standard (SP1)

Prosessori:
AMD Opteron™ Processor 6176 SE 2.30 GHz

RAM:
1,50 GB

Tyyppi:
64- bittinen käyttöjärjestelmä

Levytila:
29,8 GB

Status:
NAV-palvelin

Nimi:
config 1918VM2

Domain:
mattikatti.fi

Palvelin:
Windows Server 2008 R2 Standard (SP1)

Prosessori:
Quad-Core AMD Opteron ™ Processor 8373 2.39 GHz

RAM:
1,50 GB

Tyyppi:
64- bittinen käyttöjärjestelmä

Levytila:
29,8 GB

Status:
SQL Server 2008 -palvelin

Nimi:
config 1918VM1

Domain:
mattikatti.fi

Palvelin:
Windows 7 Enterprise (SP1)

Prosessori:
Quad-Core AMD Opteron ™ Processor 8373 2.39 GHz

RAM:
1,50 GB

Tyyppi:
32- bittinen käyttöjärjestelmä

Levytila:
23,8 GB

Status:
NAV-client

Nimi:
config 1918VM0

Domain:
mattikatti.fi

Palvelin:
Windows Server 2008 R2 Standard (SP1)

Prosessori:
Quad-Core AMD Opteron ™ Processor 8373 2.39 GHz

RAM:
1,50 GB

Tyyppi:
64-bittinen käyttöjärjestelmä

Levytila:
29,8 GB

Status:
DC-palvelin

LIITE 3

TYÖKUORMAMALLI

Työkuormamalli

Sovellus 1

Mitkä ovat sovelluksen avainskenaariot?

- Sisään kirjautuminen
- Tiedon hakeminen
- Tiedon syöttö-, muokkaus ja poisto
- Erityishaku (graafinen)
- Erityishaku (taulukko)

Millainen määrä käyttäjiä maksimissaan kirjautuu sisään kerrallaan?

Sovelluksen käyttäjämäärä on pieni. Todennäköisesti neljä profiilia (1+3). Testauksen kautta pyritään kartoittamaan käyttäjämäärän kasvua kuitenkin yli tarpeen.

Millaiset toimet käyttäjä voi sisällä toteuttaa?

peruskäyttäjä voi:

- kirjautua sisään
- navigoida toimintosivulle ja toteuttaa siellä kaikki mahdolliset asiakastietokannan hallintatoimet
- lukea asiakastietoja
- vaihtaa salasanansa
- tarkastaa graafisessa muodossa olevan asiakaslistauksen paikkakunnittain
- tarkastaa taulukkomuodossa olevan asiakaslistauksen paikkakunnittain
- kirjautua ulos

Lisäksi pääkäyttäjä voi edellä mainitun lisäksi:

- luoda käyttäjätilin ja määrittää sille roolin

Millaiset käyttäjäprofiilit sovelluksessa on käytössä?

2 erilaista profiilia: pääkäyttäjä (pk) ja peruskäyttäjä (per). Profiililla pk pääsee kaikkialle, peruskäyttäjällä vain pääkansioon.

Millaiset toimet ovat keskiverto kirjautumisen aikana toteutettavissa?

Peruskäyttäjän keskivertosessio kestää noin 5 minuuttia. Tänä aikana käyttäjä tarkastaa asiakastiedot, asiakkaan tiedot paikkakunnittain, lisää, poistaa tai muokkaa asiakastietoja. Pääkäyttäjän syy kirjautua sisään on yleensä uuden käyttäjätilin luominen.

Millainen on keskiverto viive pyyntöjen välillä?

Sovellus on pieni ja käyttöasteeltaan kevyt. Sovelluksen pyyntökuormitusmäärä ei ole suuri. Yksi käyttäjä suorittaa pyyntöön johtavan toimen noin kerran puolessa minuutissa. Tuo väliaika menee tiedon lukuun tai kirjaamiseen. Käyttäjiä on arvion mukaan kerrallaan vähän ja mitoitus on tehty kaikkiaan neljälle, joten pyyntösarjoja (get - post) ei tule kovinkaan paljoa minuuttia kohden. Jos järjestelmä kestää tuon kuormamäärän, se todennäköisesti pärjää käytössä.

Profiilisekoitus?

1+3 (pk + per)

Kuinka mittavan testauksen sovellus tarvitsee?

Sovellus on pieni, joten testiä ei ole sen puolesta syytä jakaa pienempiin osiin. Sovellukselle suoritetaan kuormitus- ja rasitustestaus. Sovellus ei ole kriittinen kokonaisen päivän päällä oleva sovellus, vaan sinne kirjaudutaan sisään, suoritetaan halutut toimet ja kirjaudutaan ulos. Tämä lisää kirjautumisresurssikuormitusta, mutta kirjautuja ei ole sisällä montaakaan minuuttia. Testin täydellinen läpivieminen ottaa noin pari päivää sisältäen lukuisia pieniä testejä, joiden aika vaihtelee viidestä minuutista kymmeneen minuuttiin. Niistä kuormitustestaus ottaa päivän ja rasitustestaus noin päivän. Lisäksi testaukselle on tarpeen varata aikaa viikon verran, jotta mahdolliset ongelmat eivät haittaa prosessia. Lopuksi suoritetaan maksimissaan 30 minuuttia kestävä rasitustestaus, jossa sovellus asetetaan poikkeuksellisen suuren rasituksen alle. Näin saadaan selville verkkohyökkäysten tai muiden poikkeustilanteiden kannalta kriittisimmät riskit.

Sovellus 2

Mitkä ovat sovelluksen avainskenaariot?

- Sisään kirjautuminen
- Tiedon hakeminen
- Tiedon syöttö
- Tiedon muokkaus
- Tiedon poisto
- Erityishaku (graafinen)
- Erityishaku (taulukko)

Millainen määrä käyttäjiä maksimissaan kirjautuu sisään kerrallaan?

Sovelluksen käyttäjämäärä on pieni. Todennäköisesti neljä profiilia (1+3). Testauksen kautta pyritään kartoittamaan käyttäjämäärän kasvua kuitenkin yli tarpeen.

Millaiset toimet käyttäjä voi sisällä toteuttaa?

peruskäyttäjä voi:

- kirjautua sisään
- lisätä asiakkaan
- poistaa tai muokata asiakastietoa
- lukea asiakastietoja
- vaihtaa salasanansa
- tarkastaa graafisessa muodossa olevan asiakaslistauksen paikkakunnittain
- tarkastaa taulukkomuodossa olevan asiakaslistauksen paikkakunnittain
- kirjautua ulos

Lisäksi pääkäyttäjä voi edellä mainitun lisäksi:

- luoda käyttäjätilin ja määrittää sille roolin

Millaiset käyttäjäprofiilit sovelluksessa on käytössä?

2 erilaista profiilia: pääkäyttäjä ja peruskäyttäjä

Millaiset toimet ovat keskiverto kirjautumisen aikana toteutettavissa?

Pohtiessa peruskäyttäjän keskivertosessiota, sen kesto on noin 5 minuuttia. Tänä aikana käyttäjä tarkastaa asiakastiedot, tiedot paikkakunnittain, lisää, poistaa tai muokkaa asiakastietoja. Pääkäyttäjän syy kirjautua sisään on yleensä uuden käyttäjätilin luominen.

Millainen on keskiverto viive pyyntöjen välillä?

Sovellus on pieni ja käyttöasteeltaan kevyt. Sovelluksen pyyntökuormitusmäärä ei ole suuri. Yksi käyttäjä suorittaa pyyntöön johtavan toimen noin kerran puolessa minuutissa. Tuo väliaika menee tiedon lukuun tai kirjaamiseen. Käyttäjiä on arvion mukaan kerrallaan vähän ja mitoitus on tehty kaikkiaan neljälle, joten pyyntösarjoja (get - post) ei tule kovinkaan paljoa minuuttia kohden. Jos järjestelmä kestää tuon kuormamäärän, se todennäköisesti pärjää käytössä.

Profiilisekoitus?

1+3

Kuinka mittavan testauksen sovellus tarvitsee?

Sovellus on pieni, joten testiä ei ole sen puolesta syytä jakaa pienempiin osiin. Sovellukselle suoritetaan kuormitus- ja rasitustestaus. Sovellus ei ole kriittinen kokonaisen päivän päällä oleva sovellus, vaan sinne kirjaudutaan sisään, suoritetaan halutut toimet ja kirjaudutaan ulos. Tämä lisää kirjautumisresurssikuormitusta, mutta kirjautuja ei ole sisällä montaakaan minuuttia. Testin täydellinen läpivieminen ottaa noin pari päivää sisältäen lukuisia pieniä testejä, joiden aika vaihtelee viidestä minuutista kymmeneen minuuttiin. Niistä kuormitustestaus ottaa päivän ja rasitustestaus noin päivän. Lisäksi testaukselle on tarpeen varata aikaa viikon verran, jotta mahdolliset ongelmat eivät haittaa prosessia. Lopuksi suoritetaan maksimissaan 30 minuuttia kestävä rasitustestaus, jossa sovellus asetetaan poikkeuksellisen suuren rasituksen alle. Näin saadaan selville verkkohyökkäysten tai muiden poikkeustilanteiden kannalta kriittisimmät riskit.

LIITE 4

STORED PROCEDURES, ASIAKASINPUT

```
ALTER PROCEDURE asiakasinput
(
@nimi varchar(100),
@paikkakunta varchar(100),
@osoite varchar(100),
@postinumero varchar(100),
@puhelin varchar(100)
)
AS
insert into asiakas (nimi, paikkakunta, osoite, postinumero, puhelin)
values (@nimi, @paikkakunta, @osoite, @postinumero, @puhelin)
```

LIITE 5

STORED PROCEDURES, PAIKKAKUNNITTAIN

```
ALTER PROCEDURE paikkakunnittain
as
Select  paikkakunta, count (paikkakunta) as "asiakkaat" from asiakas
group by paikkakunta
```

LIITE 6

STORED PROCEDURES, PERUSSELECT

```
ALTER PROCEDURE perusselect
AS
Select * From asiakas
```

LIITE 7

LINQ, PERUSHAKU

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Load
        Dim dc As New DataClassesAsiakasDataContext()
        Dim query = From m In dc.asiakas Select m
        Me.GridView1.DataSource = query
        Me.GridView1.DataBind()
End Sub
```

LIITE 8

LINQ, ASIAKKAIDEN LUKUMÄÄRÄ PAIKKAKUNNITTAIN (KAKSI TAPAA)

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
    'graafinen esitys
    Dim dc As New DataClassesAsiakasDataContext()
    Dim query = From m In dc.asiakas Group By m.paikkakunta Into g = Group, Count()
    Me.BulletedList1.DataSource = query
    Me.BulletedList1.DataBind()
End Sub
```

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Load
    'taulukkomuotoinen esitys
    Dim dc As New DataClassesAsiakasDataContext()
    Dim query = From m In dc.asiakas Group By m.paikkakunta Into g = Group, Count()
    Me.GridView1.DataSource = query
    Me.GridView1.DataBind()
End Sub
```

LIITE 9

TIEDON SYÖTTÄMINEN (STORED PROCEDURES)

```
Protected Sub Button1_Click(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Button1.Click

 'määritetään arvot
 'muuttujan arvo - vastaava tekstikenttä sivulla

        Dim nimi As String = txtNimi.Text
        Dim osoite As String = txtOsoite.Text
        Dim paikkakunta As String = txtPaikkakunta.Text
        Dim postinumero As String = txtPstinro.Text
        Dim puhelin As String = txtPuh.Text


        ' tiedot tietokantaan

'määritetään tekstikentän tieto siirtyy oikeaan kohtaan tietokannassa
'määritetään yksilöivä connectionstring

        Dim datas As New SqlDataSource()
        datas.ConnectionString =
ConfigurationManager.ConnectionStrings("asiakasConnectionString").ToSt
ring()

'stored proceduren (nimeltään tiedonsyotto)käyttö
        datas.InsertCommandType =
SqlDataSourceCommandType.StoredProcedure
        datas.InsertCommand = "asiakasinput"
'data oikeille paikoilleen
        datas.InsertParameters.Add("nimi", nimi)
        datas.InsertParameters.Add("osoite", osoite)
        datas.InsertParameters.Add("paikkakunta", paikkakunta)
        datas.InsertParameters.Add("postinumero", postinumero)
        datas.InsertParameters.Add("puhelin", puhelin)


'aloitetaan virhekäsittelyt

        Dim rowsaffected As Integer = 0

        Try
            rowsaffected = datas.Insert()
        Catch ex As Exception
            Response.Write(ex.Message.ToString)
            Exit Sub
        Finally
            datas = Nothing
        End Try

        If rowsaffected <> 1 Then

            Response.Write("error")
        End If
        Exit Sub
    END SUB
```

LIITE 10

WEB.CONFIG

```
<?xml version="1.0" encoding="utf-8" ?>
<siteMap xmlns="http://schemas.microsoft.com/AspNet/SiteMap-File-1.0"
>
    <siteMapNode url="" title=""  description="">
        <siteMapNode url="cpass.aspx" title="salasana"  description=""
/>
        <siteMapNode url="readdata.aspx" title="lue"  description=""
/>
        <siteMapNode url="inputdata.aspx" title="syötä"
description="" />
        <siteMapNode url="modifydeletedata.aspx"
title="muokkaa/poista"  description="" />
      <siteMapNode url="gridstatistics.aspx" title="taulukko stat"
description="" />
      <siteMapNode url="statistics.aspx" title="graafinen stat"
description="" />
      <siteMapNode url="control/Default.aspx" title="pääkäyttäjä"
description="" />

    </siteMapNode>
</siteMap>
```

LIITE 11

SESSIO

```
End Sub

    Protected Sub Page_Load(ByVal sender As Object, ByVal e As
System.EventArgs) Handles Me.Load
        ' luodaan yksilöllinen numeroarvo tekstikenttään
        Dim code = 1
        Dim rand = Rnd(code)
        txtRandom.Text = rand

        'Luodaan sessio
        Session("mykey") = txtRandom.Text
        Dim myValue As String = CType(Session("mykey"), String)


    End Sub
```